Clang Documentation

version 15

The Clang Team

Contents

Clang pre-release 15 Release Notes 1

- Introduction 2
- What's New in Clang pre-release 15? 2
- Potentially Breaking Changes 2
  - Major New Features 2
  - Bug Fixes 2
    - Improvements to Clang's diagnostics 4
  - Non-comprehensive list of changes in this release 5
  - New Compiler Flags 5
  - Deprecated Compiler Flags 5
  - Modified Compiler Flags 5
  - Removed Compiler Flags 5
  - New Pragmas in Clang 5
  - Attribute Changes in Clang 6
  - Windows Support 6
  - AIX Support 6
  - C Language Changes in Clang 6
  - C2x Feature Support 6
  - C++ Language Changes in Clang 7
    - C++20 Feature Support 7
    - C++2b Feature Support 7
  - CUDA/HIP Language Changes in Clang 7
  - Objective-C Language Changes in Clang 8
  - OpenCL C Language Changes in Clang 8
  - ABI Changes in Clang 8
  - OpenMP Support in Clang 8
  - CUDA Support in Clang 8
  - X86 Support in Clang 8
  - DWARF Support in Clang 8
  - Arm and AArch64 Support in Clang 8
  - Floating Point Support in Clang 8
  - Internal API Changes 8
  - Build System Changes 8
  - AST Matchers 8
  - clang-format 8
  - libclang 9
  - Static Analyzer 9
  - Undefined Behavior Sanitizer (UBSan) 9
- Core Analysis Improvements 9
- New Issues Found 9
  - Python Binding Changes 9
- Significant Known Problems 9
- Additional Information 9

Using Clang as a Compiler 10

Clang Compiler User's Manual 10
Introduction 12
Terminology 12
Basic Usage 12
Command Line Options 13
Options to Control Error and Warning Messages 13
Formatting of Diagnostics 13
Individual Warning Groups 17
Options to Control Clang Crash Diagnostics 18
Options to Emit Optimization Reports 18
Current limitations 19
Options to Emit Resource Consumption Reports 19
Other Options 20
Configuration files 20
Language and Target-Independent Features 21
Controlling Errors and Warnings 21
Controlling How Clang Displays Diagnostics 21
Diagnostic Mappings 22
Diagnostic Categories 22
Controlling Diagnostics via Command Line Flags 22
Controlling Diagnostics via Pragmas 22
Controlling Diagnostics in System Headers 23
Controlling Deprecation Diagnostics in Clang-Provided C Runtime Headers 23
Enabling All Diagnostics 24
Controlling Static Analyzer Diagnostics 24
Precompiled Headers 24
Generating a PCH File 24
Using a PCH File 24
Relocatable PCH Files 25
Controlling Floating Point Behavior 25
A note about `__FLT_EVAL_METHOD__` 29
A note about Floating Point Constant Evaluation 29
Controlling Code Generation 30
Profile Guided Optimization 33
Differences Between Sampling and Instrumentation 34
Using Sampling Profilers 34
Sample Profile Formats 34
Sample Profile Text Format 35
Profiling with Instrumentation 36
Disabling Instrumentation 38
Instrumenting only selected files or functions 38
Profile remapping 39
GCOV-based Profiling 40
Controlling Debug Information 40
Controlling Size of Debug Information 40
Controlling Macro Debug Info Generation 41

Controlling Debugger “Tuning” 41
Controlling LLVM IR Output 41
Controlling Value Names in LLVM IR 41
Comment Parsing Options 42
C Language Features 42
Extensions supported by clang 42
Differences between various standard modes 42
GCC extensions not implemented yet 43
Intentionally unsupported GCC extensions 43
Microsoft extensions 44
C++ Language Features 44
Controlling implementation limits 44
Objective-C Language Features 44
Objective-C++ Language Features 44
OpenMP Features 44
Controlling implementation limits 45
OpenCL Features 45
OpenCL Specific Options 45
OpenCL Targets 46
Specific Targets 46
Generic Targets 46
OpenCL Header 47
OpenCL Extensions 47
OpenCL-Specific Attributes 47
nosvm 47
opencl_unroll_hint 48
convergent 48
noduplicate 48
C++ for OpenCL 49
Constructing and destroying global objects 49
Libraries 50
Target-Specific Features and Limitations 50
CPU Architectures Features and Limitations 50
X86 50
ARM 50
PowerPC 50
Other platforms 50
Operating System Features and Limitations 51
Windows 51
Cygwin 51
MinGW32 51
MinGW-w64 51
AIX 51
SPIR-V support 52
clang-cl 52
Command-Line Options 53

The /clang: Option 57
The /Zc:dllexportInlines- Option 57
Finding Clang runtime libraries 58
Assembling a Complete Toolchain 59
Introduction 59
Tools 60
Clang frontend 60
Language frontends for other languages 60
Assembler 60
Linker 60
Runtime libraries 61
Compiler runtime 61
compiler-rt (LLVM) 61
libgcc_s (GNU) 61
Atomics library 61
compiler-rt (LLVM) 61
libatomic (GNU) 62
Unwind library 62
libunwind (LLVM) 62
libgcc_s (GNU) 62
libunwind (nongnu.org) 62
libunwind (PathScale) 62
Sanitizer runtime 62
C standard library 62
C++ ABI library 62
libc++abi (LLVM) 63
libsupc++ (GNU) 63
libcxxrt (PathScale) 63
C++ standard library 63
libc++ (LLVM) 63
libstdc++ (GNU) 63
Clang Language Extensions 64
Objective-C Literals 65
Introduction 65
NSNumber Literals 65
Examples 65
Discussion 65
Boxed Expressions 66
Boxed Enums 66
Boxed C Strings 67
Boxed C Structures 67
Container Literals 68
Examples 68
Discussion 68
Object Subscripting 68
Examples 68

Subscripting Methods 69
Array-Style Subscripting 69
Dictionary-Style Subscripting 69
Discussion 70
Caveats 70
Grammar Additions 70
Availability Checks 71
Language Specification for Blocks 72
Revisions 72
Overview 72
The Block Type 72
Block Variable Declarations 72
Block Literal Expressions 73
The Invoke Operator 74
The Copy and Release Operations 74
The `__block` Storage Qualifier 74
Control Flow 74
Objective-C Extensions 74
C++ Extensions 75
Block Implementation Specification 76
History 76
High Level 77
Imported Variables 78
Imported `const` copy variables 79
Imported `const` copy of `Block` reference 79
Importing `__attribute__((NSObject))` variables 80
Imported `__block` marked variables 81
Layout of `__block` marked variables 81
Access to `__block` variables from within its lexical scope 81
Importing `__block` variables into `Blocks` 82
Importing `__attribute__((NSObject))` `__block` variables 83
`__block` escapes 83
Nesting 83
Objective C Extensions to `Blocks` 83
Importing Objects 83
`Blocks` as Objects 83
`__weak` `__block` Support 84
C++ Support 85
Runtime Helper Functions 86
Copyright 87
Objective-C Automatic Reference Counting (ARC) 88
About this document 89
Purpose 89
Background 89
Evolution 90
General 91

Retainable object pointers 91
Retain count semantics 92
Retainable object pointers as operands and arguments 92
Consumed parameters 93
Retained return values 93
Unretained return values 94
Bridged casts 95
Restrictions 95
Conversion of retainable object pointers 95
Conversion to retainable object pointer type of expressions with known semantics 95
Conversion from retainable object pointer type in certain contexts 96
Ownership qualification 96
Spelling 97
Property declarations 98
Semantics 98
Restrictions 99
Weak-unavailable types 99
Storage duration of `__autoreleasing` objects 99
Conversion of pointers to ownership-qualified types 100
Passing to an out parameter by writeback 101
Ownership-qualified fields of structs and unions 102
Formal rules for non-trivial types in C 102
Application of the formal C rules to nontrivial ownership qualifiers 104
C/C++ compatibility for structs and unions with non-trivial members 104
Ownership inference 105
Objects 105
Indirect parameters 105
Template arguments 106
Method families 106
Explicit method family control 107
Semantics of method families 107
Semantics of `init` 107
Related result types 108
Optimization 108
Object liveness 109
No object lifetime extension 110
Precise lifetime semantics 110
Miscellaneous 111
Special methods 111
Memory management methods 111
`dealloc` 111
`@autoreleasepool` 112
Externally-Retained Variables 113
`self` 113
Fast enumeration iteration variables 114
Blocks 114

Exceptions 114
Interior pointers 115
C retainable pointer types 115
Auditing of C retainable pointer interfaces 116
Runtime support 116
`id objc_autorelease(id value);` 117
`void objc_autoreleasePoolPop(void *pool);` 117
`void *objc_autoreleasePoolPush(void);` 117
`id objc_autoreleaseReturnValue(id value);` 117
`void objc_copyWeak(id *dest, id *src);` 118
`void objc_destroyWeak(id *object);` 118
`id objc_initWeak(id *object, id value);` 118
`id objc_loadWeak(id *object);` 118
`id objc_loadWeakRetained(id *object);` 119
`void objc_moveWeak(id *dest, id *src);` 119
`void objc_release(id value);` 119
`id objc_retain(id value);` 119
`id objc_retainAutorelease(id value);` 119
`id objc_retainAutoreleaseReturnValue(id value);` 119
`id objc_retainAutoreleasedReturnValue(id value);` 119
`id objc_retainBlock(id value);` 120
`void objc_storeStrong(id *object, id value);` 120
`id objc_storeWeak(id *object, id value);` 120
`id objc_unsafeClaimAutoreleasedReturnValue(id value);` 120
Matrix Types 121
Draft Specification 121
Matrix Type 121
Matrix Type Attribute 121
Standard Conversions 121
Arithmetic Conversions 122
Matrix Type Element Access Operator 122
Matrix Type Binary Operators 122
Matrix Type Builtin Operations 123
TODOs 124
Decisions for the Implementation in Clang 124
Introduction 124
Feature Checking Macros 124
`__has_builtin` 124
`__has_feature` and `__has_extension` 125
`__has_cpp_attribute` 126
`__has_c_attribute` 126
`__has_attribute` 126
`__has_declspec_attribute` 127
`__is_identifier` 127
Include File Checking Macros 127
`__has_include` 127

__has_include_next 128
__has_warning 128
Builtin Macros 128
Vectors and Extended Vectors 129
Boolean Vectors 130
Vector Literals 130
Vector Operations 131
Vector Builtins 131
Matrix Types 133
Half-Precision Floating Point 134
Messages on deprecated and unavailable Attributes 135
Attributes on Enumerators 135
C++11 Attributes on using-declarations 135
'User-Specified' System Frameworks 135
Checks for Standard Language Features 136
C++98 136
C++ exceptions 136
C++ RTTI 136
C++11 136
C++11 SFINAE includes access control 136
C++11 alias templates 136
C++11 alignment specifiers 136
C++11 attributes 136
C++11 generalized constant expressions 136
C++11 decltype() 137
C++11 default template arguments in function templates 137
C++11 defaulted functions 137
C++11 delegating constructors 137
C++11 deleted functions 137
C++11 explicit conversion functions 137
C++11 generalized initializers 137
C++11 implicit move constructors/assignment operators 137
C++11 inheriting constructors 137
C++11 inline namespaces 137
C++11 lambdas 137
C++11 local and unnamed types as template arguments 138
C++11 noexcept 138
C++11 in-class non-static data member initialization 138
C++11 nullptr 138
C++11 override control 138
C++11 reference-qualified functions 138
C++11 range-based for loop 138
C++11 raw string literals 138
C++11 rvalue references 138
C++11 static_assert() 138
C++11 thread_local 138

C++11 type inference 138
C++11 strongly typed enumerations 139
C++11 trailing return type 139
C++11 Unicode string literals 139
C++11 unrestricted unions 139
C++11 user-defined literals 139
C++11 variadic templates 139
C++14 139
C++14 binary literals 139
C++14 contextual conversions 139
C++14 decltype(auto) 139
C++14 default initializers for aggregates 139
C++14 digit separators 139
C++14 generalized lambda capture 140
C++14 generic lambdas 140
C++14 relaxed constexpr 140
C++14 return type deduction 140
C++14 runtime-sized arrays 140
C++14 variable templates 140
C11 140
C11 alignment specifiers 140
C11 atomic operations 140
C11 generic selections 141
C11 `_Static_assert()` 141
C11 `_Thread_local` 141
Modules 141
Type Trait Primitives 141
Blocks 144
ASM Goto with Output Constraints 144
Objective-C Features 145
Related result types 145
Automatic reference counting 146
Weak references 146
Enumerations with a fixed underlying type 147
Interoperability with C++11 lambdas 147
Object Literals and Subscripting 147
Objective-C Autosynthesis of Properties 148
Objective-C retaining behavior attributes 148
Objective-C @available 148
Objective-C++ ABI: protocol-qualifier mangling of parameters 149
Initializer lists for complex numbers in C 149
OpenCL Features 150
`__cl_clang_bitfields` 150
`__cl_clang_function_pointers` 150
`__cl_clang_variadic_functions` 151
`__cl_clang_non_portable_kernel_param_types` 151

Remove address space builtin function 152
Legacy 1.x atomics with generic address space 152
Builtin Functions 152
__builtin_alloca 152
__builtin_alloca_with_align 153
__builtin_assume 153
__builtin_call_with_static_chain 154
__builtin_readcyclecounter 154
__builtin_dump_struct 154
__builtin_shufflevector 155
__builtin_convertvector 156
__builtin_bitreverse 156
__builtin_rotateleft 157
__builtin_rotateright 157
__builtin_unreachable 158
__builtin_unpredictable 158
__builtin_expect 158
__builtin_expect_with_probability 159
__builtin_prefetch 159
__sync_swap 159
__builtin_addressof 160
__builtin_function_start 160
__builtin_operator_new and __builtin_operator_delete 160
__builtin_preserve_access_index 161
__builtin_debugtrap 161
__builtin_trap 161
__builtin_sycl_unique_stable_name 162
Multiprecision Arithmetic Builtins 162
Checked Arithmetic Builtins 162
Floating point builtins 163
__builtin_canonicalize 163
String builtins 163
Memory builtins 164
Guaranteed inlined copy 164
Atomic Min/Max builtins with memory ordering 164
__c11_atomic builtins 165
Low-level ARM exclusive memory builtins 165
Non-temporal load/store builtins 166
C++ Coroutines support builtins 166
Source location builtins 167
Alignment builtins 168
Non-standard C++11 Attributes 169
Target-Specific Extensions 169
ARM/AArch64 Language Extensions 169
Memory Barrier Intrinsics 169
X86/X86-64 Language Extensions 169

Memory references to specified segments 169
PowerPC Language Extensions 170
Set the Floating Point Rounding Mode 170
PowerPC cache builtins 170
Extensions for Static Analysis 170
Extensions for Dynamic Analysis 170
Extensions for selectively disabling optimization 171
Extensions for loop hint optimizations 172
Vectorization, Interleaving, and Predication 172
Loop Unrolling 173
Loop Distribution 173
Additional Information 174
Extensions to specify floating-point flags 174
Specifying an attribute for multiple declarations (#pragma clang attribute) 176
Subject Match Rules 177
Supported Attributes 178
Specifying section names for global objects (#pragma clang section) 178
Specifying Linker Options on ELF Targets 179
Evaluating Object Size Dynamically 179
Deprecating Macros 179
Restricted Expansion Macros 179
Final Macros 180
Line Control 180
Extended Integer Types 181
Intrinsics Support within Constant Expressions 181
Clang command line argument reference 183
Introduction 184
Actions 193
Compilation flags 194
Preprocessor flags 197
Include path management 198
Dependency file generation 200
Dumping preprocessor state 200
Diagnostic flags 201
Target-independent compilation options 201
OpenCL flags 219
SYCL flags 220
Target-dependent compilation options 220
AARCH64 226
AMDGPU 227
ARM 227
Hexagon 228
Hexagon 228
M68k 228
MIPS 229
PowerPC 230

WebAssembly 231
WebAssembly Driver 231
X86 231
RISCV 233
Long double flags 234
Optimization level 234
Debug information generation 234
Kind and level of debug information 234
Debug level 234
Debugger to tune debug information for 235
Debug information flags 235
Static analyzer flags 235
Fortran compilation flags 235
Linker flags 237
<clang-dxc options> 238
Attributes in Clang 239
Introduction 244
AMD GPU Attributes 245
amdgpu_flat_work_group_size 245
amdgpu_num_sgpr 245
amdgpu_num_vgpr 246
amdgpu_waves_per_eu 246
Calling Conventions 247
aarch64_sve_pcs 247
aarch64_vector_pcs 247
fastcall 248
ms_abi 248
pcs 248
preserve_all 249
preserve_most 249
regcall 250
regparm 250
stdcall 250
thiscall 250
vectorcall 251
Consumed Annotation Checking 251
callable_when 251
consumable 252
param_typestate 252
return_typestate 252
set_typestate 252
test_typestate 253
Customizing Swift Import 253
swift_async 253
swift_async_error 254
swift_async_name 254

swift_attr 255
swift_bridge 255
swift_bridged 255
swift_error 256
swift_name 256
swift_newtype 257
swift_objc_members 257
swift_private 257
Declaration Attributes 258
Owner 258
Pointer 258
_Packed 259
__single_inhertiance, __multiple_inheritance, __virtual_inheritance 260
asm 260
deprecated 261
empty_bases 261
enum_extensibility 262
external_source_symbol 263
flag_enum 263
layout_version 264
lto_visibility_public 264
managed 264
novtable 264
ns_error_domain 265
objc_boxable 265
objc_direct 266
objc_direct_members 267
objc_non_runtime_protocol 267
objc_nonlazy_class 268
objc_runtime_name 268
objc_runtime_visible 268
objc_subclassing_restricted 269
preferred_name 269
randomize_layout, no_randomize_layout 269
randomize_layout, no_randomize_layout 270
selectany 270
transparent_union 270
trivial_abi 271
using_if_exists 272
Field Attributes 272
no_unique_address 272
Function Attributes 273
#pragma omp declare simd 273
#pragma omp declare target 273
#pragma omp declare variant 274
SV_GroupIndex 275

_Export 275
_Noreturn 275
abi_tag 275
acquire_capability, acquire_shared_capability 276
alloc_align 276
alloc_size 277
allocator 277
always_inline, __force_inline 278
artificial 278
assert_capability, assert_shared_capability 279
assume 279
assume_aligned 279
availability 280
btf_decl_tag 282
callback 282
carries_dependency 283
cf_consumed 283
cf_returns_not_retained 284
cf_returns_retained 285
cfi_canonical_jump_table 286
clang::builtin_alias, clang_builtin_alias 286
clang_arm_builtin_alias 287
cmse_nonsecure_entry 287
code_seg 287
convergent 288
cpu_dispatch 288
cpu_specific 289
diagnose_as_builtin 290
diagnose_if 291
disable_sanitizer_instrumentation 292
disable_tail_calls 292
enable_if 293
enforce_tcb 295
enforce_tcb_leaf 295
error, warning 295
exclude_from_explicit_instantiation 296
export_name 297
flatten 297
force_align_arg_pointer 297
format 298
gnu_inline 299
guard 299
ifunc 300
import_module 300
import_name 300
internal_linkage 301

interrupt (ARM) 301
interrupt (AVR) 302
interrupt (MIPS) 302
interrupt (RISCV) 302
kernel 303
lifetimebound 303
long_call, far 304
malloc 304
micromips 305
mig_server_routine 305
min_vector_width 305
no_builtin 306
no_caller_saved_registers 306
no_profile_instrument_function 307
no_sanitize 307
no_sanitize_address, no_address_safety_analysis 308
no_sanitize_memory 308
no_sanitize_thread 308
no_speculative_load_hardening 309
no_split_stack 310
no_stack_protector 310
noalias 310
nocf_check 310
nodiscard, warn_unused_result 311
noduplicate 311
noinline 312
nomicromips 312
noreturn, _Noreturn 313
not_tail_called 313
nothrow 314
ns_consumed 314
ns_consumes_self 315
ns_returns_autoreleased 316
ns_returns_not_retained 317
ns_returns_retained 317
numthreads 318
objc_method_family 319
objc_requires_super 319
optnone 320
os_consumed 320
os_consumes_this 321
os_returns_not_retained 322
os_returns_retained 322
os_returns_retained_on_non_zero 323
os_returns_retained_on_zero 324
overloadable 325

patchable_function_entry 326
preserve_access_index 327
reinitializes 327
release_capability, release_shared_capability 328
retain 328
shader 328
short_call, near 329
signal 329
speculative_load_hardening 329
sycl_kernel 330
target 331
target_clones 332
try_acquire_capability, try_acquire_shared_capability 332
used 333
xray_always_instrument, xray_never_instrument, xray_log_args 333
xray_always_instrument, xray_never_instrument, xray_log_args 334
zero_call_used_regs 334
Handle Attributes 335
acquire_handle 335
release_handle 335
use_handle 335
Nullability Attributes 336
_Nonnull 336
_Null_unspecified 337
_Nullable 337
_Nullable_result 337
nonnull 338
returns_nonnull 338
OpenCL Address Spaces 339
[[clang::opencl_global_device]], [[clang::opencl_global_host]] 339
[[clang::opencl_global_device]], [[clang::opencl_global_host]] 339
__constant, constant, [[clang::opencl_constant]] 340
__generic, generic, [[clang::opencl_generic]] 340
__global, global, [[clang::opencl_global]] 340
__local, local, [[clang::opencl_local]] 341
__private, private, [[clang::opencl_private]] 341
Statement Attributes 341
#pragma clang loop 341
#pragma unroll, #pragma nounroll 342
__read_only, __write_only, __read_write (read_only, write_only, read_write) 343
fallthrough 344
intel_reqd_sub_group_size 345
likely and unlikely 345
likely and unlikely 347
musttail 350
nomerge 350

opencl_unroll_hint 351
suppress 351
sycl_special_class 351
Type Attributes 352
__ptr32 352
__ptr64 353
__sptr 353
__uptr 353
align_value 353
arm_sve_vector_bits 354
btf_type_tag 354
clang_arm_mve_strict_polymorphism 354
cmse_nonsecure_call 355
device_builtin_surface_type 355
device_builtin_texture_type 356
noderef 356
objc_class_stub 357
Type Safety Checking 357
argument_with_type_tag 358
pointer_with_type_tag 358
type_tag_for_datatype 359
Variable Attributes 361
always_destroy 361
called_once 361
dllexport 362
dllimport 362
init_priority 363
init_seg 363
leaf 363
loader_uninitialized 364
maybe_unused, unused 364
no_destroy 364
nodebug 365
noescape 365
nosvm 366
objc_externally_retained 366
pass_object_size, pass_dynamic_object_size 367
require_constant_initialization, constinit (C++20) 369
section, __declspec(allocate) 369
standalone_debug 370
swift_async_context 370
swift_context 370
swift_error_result 370
swift_indirect_result 371
swiftasynccall 372
swiftcall 372

thread 373
tls_model 373
uninitialized 373
Diagnostic flags in Clang 375
Introduction 396
Diagnostic flags 396
-W 396
-W#pragma-messages 396
-W#warnings 396
-WCFString-literal 396
-WCL4 396
-WIndependentClass-attribute 396
-WNSObject-attribute 396
-Wabi 397
-Wabsolute-value 397
-Wabstract-final-class 397
-Wabstract-vbase-init 397
-Waddress 397
-Waddress-of-packed-member 397
-Waddress-of-temporary 397
-Waggregate-return 397
-Waggressive-restrict 397
-Waix-compat 398
-Walign-mismatch 398
-Wall 398
-Walloca 398
-Walloca-with-align-alignof 398
-Walways-inline-coroutine 398
-Wambiguous-delete 398
-Wambiguous-ellipsis 398
-Wambiguous-macro 399
-Wambiguous-member-template 399
-Wambiguous-reversed-operator 399
-Wanalyzer-incompatible-plugin 399
-Wanon-enum-enum-conversion 399
-Wanonymous-pack-parens 399
-Warc 399
-Warc-bridge-casts-disallowed-in-nonarc 399
-Warc-maybe-repeated-use-of-weak 400
-Warc-non-pod-memaccess 400
-Warc-performSelector-leaks 400
-Warc-repeated-use-of-weak 400
-Warc-retain-cycles 400
-Warc-unsafe-retained-assign 400
-Wargument-outside-range 400
-Wargument-undefined-behaviour 400

-Warray-bounds 401
-Warray-bounds-pointer-arithmetic 401
-Wasm 401
-Wasm-operand-widths 401
-Wassign-enum 401
-Wassume 401
-Wat-protocol 401
-Watimport-in-framework-header 401
-Watomic-access 402
-Watomic-alignment 402
-Watomic-implicit-seq-cst 402
-Watomic-memory-ordering 402
-Watomic-properties 402
-Watomic-property-with-user-defined-accessor 402
-Wattribute-packed-for-bitfield 402
-Wattribute-warning 402
-Wattributes 403
-Wauto-disable-vptr-sanitizer 403
-Wauto-import 403
-Wauto-storage-class 403
-Wauto-var-id 403
-Wavailability 403
-Wavr-rtlib-linking-quirks 404
-Wbackend-plugin 404
-Wbackslash-newline-escape 404
-Wbad-function-cast 404
-Wbinary-literal 404
-Wbind-to-temporary-copy 404
-Wbinding-in-condition 404
-Wbit-int-extension 405
-Wbitfield-constant-conversion 405
-Wbitfield-enum-conversion 405
-Wbitfield-width 405
-Wbitwise-conditional-parentheses 405
-Wbitwise-instead-of-logical 405
-Wbitwise-op-parentheses 405
-Wblock-capture-autoreleasing 405
-Wbool-conversion 405
-Wbool-conversions 406
-Wbool-operation 406
-Wbraced-scalar-init 406
-Wbranch-protection 406
-Wbridge-cast 406
-Wbuiltin-assume-aligned-alignment 406
-Wbuiltin-macro-redefined 406
-Wbuiltin-memcpy-chk-size 407

-Wbuiltin-requires-header 407
-Wc++-compat 407
-Wc++0x-compat 407
-Wc++0x-extensions 407
-Wc++0x-narrowing 407
-Wc++11-compat 407
-Wc++11-compat-deprecated-writable-strings 408
-Wc++11-compat-pedantic 408
-Wc++11-compat-reserved-user-defined-literal 408
-Wc++11-extensions 408
-Wc++11-extra-semi 409
-Wc++11-inline-namespace 409
-Wc++11-long-long 409
-Wc++11-narrowing 409
-Wc++14-attribute-extensions 410
-Wc++14-binary-literal 410
-Wc++14-compat 410
-Wc++14-compat-pedantic 410
-Wc++14-extensions 410
-Wc++17-attribute-extensions 410
-Wc++17-compat 410
-Wc++17-compat-mangling 410
-Wc++17-compat-pedantic 411
-Wc++17-extensions 411
-Wc++1y-extensions 411
-Wc++1z-compat 411
-Wc++1z-compat-mangling 411
-Wc++1z-extensions 412
-Wc++20-attribute-extensions 412
-Wc++20-compat 412
-Wc++20-compat-pedantic 412
-Wc++20-designator 412
-Wc++20-extensions 412
-Wc++2a-compat 413
-Wc++2a-compat-pedantic 413
-Wc++2a-extensions 413
-Wc++2b-extensions 413
-Wc++98-c++11-c++14-c++17-compat 414
-Wc++98-c++11-c++14-c++17-compat-pedantic 414
-Wc++98-c++11-c++14-compat 414
-Wc++98-c++11-c++14-compat-pedantic 414
-Wc++98-c++11-compat 414
-Wc++98-c++11-compat-binary-literal 414
-Wc++98-c++11-compat-pedantic 414
-Wc++98-compat 414
-Wc++98-compat-bind-to-temporary-copy 416

-Wc++98-compat-extra-semi 416
-Wc++98-compat-local-type-template-args 416
-Wc++98-compat-pedantic 416
-Wc++98-compat-unnamed-type-template-args 417
-Wc11-extensions 417
-Wc2x-extensions 417
-Wc99-compat 417
-Wc99-designator 418
-Wc99-extensions 418
-Wcall-to-pure-virtual-from-ctor-dtor 418
-Wcalled-once-parameter 418
-Wcast-align 419
-Wcast-calling-convention 419
-Wcast-function-type 419
-Wcast-of-sel-type 419
-Wcast-qual 419
-Wcast-qual-unrelated 419
-Wchar-align 419
-Wchar-subscripts 419
-Wclang-cl-pch 420
-Wclass-conversion 420
-Wclass-varargs 420
-Wcmse-union-leak 420
-Wcomma 420
-Wcomment 420
-Wcomments 421
-Wcompare-distinct-pointer-types 421
-Wcompletion-handler 421
-Wcomplex-component-init 421
-Wcompound-token-split 421
-Wcompound-token-split-by-macro 421
-Wcompound-token-split-by-space 421
-Wconcepts-ts-compat 421
-Wconditional-type-mismatch 421
-Wconditional-uninitialized 422
-Wconfig-macros 422
-Wconstant-conversion 422
-Wconstant-evaluated 422
-Wconstant-logical-operand 422
-Wconstexpr-not-const 422
-Wconsumed 422
-Wconversion 423
-Wconversion-null 423
-Wcoroutine 423
-Wcoroutine-missing-unhandled-exception 423
-Wcovered-switch-default 423

-Wcpp 423
-Wcstring-format-directive 423
-Wctad-maybe-unsupported 424
-Wctor-dtor-privacy 424
-Wctu 424
-Wcuda-compat 424
-Wcustom-atomic-properties 424
-Wcxx-attribute-extension 424
-Wdangling 424
-Wdangling-else 424
-Wdangling-field 425
-Wdangling-gsl 425
-Wdangling-initializer-list 425
-Wdarwin-sdk-settings 425
-Wdate-time 425
-Wdealloc-in-category 425
-Wdebug-compression-unavailable 425
-Wdeclaration-after-statement 426
-Wdefaulted-function-deleted 426
-Wdelegating-ctor-cycles 426
-Wdelete-abstract-non-virtual-dtor 426
-Wdelete-incomplete 426
-Wdelete-non-abstract-non-virtual-dtor 426
-Wdelete-non-virtual-dtor 426
-Wdelimited-escape-sequence-extension 426
-Wdeprecate-lax-vec-conv-all 427
-Wdeprecated 427
-Wdeprecated-altivec-src-compat 427
-Wdeprecated-anon-enum-enum-conversion 427
-Wdeprecated-array-compare 427
-Wdeprecated-attributes 428
-Wdeprecated-comma-subscript 428
-Wdeprecated-copy 428
-Wdeprecated-copy-dtor 428
-Wdeprecated-copy-with-dtor 428
-Wdeprecated-copy-with-user-provided-copy 428
-Wdeprecated-copy-with-user-provided-dtor 428
-Wdeprecated-coroutine 428
-Wdeprecated-declarations 428
-Wdeprecated-dynamic-exception-spec 429
-Wdeprecated-enum-compare 429
-Wdeprecated-enum-compare-conditional 429
-Wdeprecated-enum-enum-conversion 429
-Wdeprecated-enum-float-conversion 429
-Wdeprecated-experimental-coroutine 429
-Wdeprecated-implementations 429

-Wdeprecated-increment-bool 430
-Wdeprecated-non-prototype 430
-Wdeprecated-objc-isa-usage 430
-Wdeprecated-objc-pointer-introspection 430
-Wdeprecated-objc-pointer-introspection-performSelector 430
-Wdeprecated-pragma 430
-Wdeprecated-register 430
-Wdeprecated-this-capture 431
-Wdeprecated-type 431
-Wdeprecated-volatile 431
-Wdeprecated-writable-strings 431
-Wdeprecated-xl-loop-pragmas 431
-Wdirect-ivar-access 431
-Wdisabled-macro-expansion 431
-Wdisabled-optimization 431
-Wdiscard-qual 432
-Wdistributed-object-modifiers 432
-Wdiv-by-zero 432
-Wdivision-by-zero 432
-Wdll-attribute-on-redeclaration 432
-Wdllexport-explicit-instantiation-decl 432
-Wdllimport-static-field-def 432
-Wdocumentation 432
-Wdocumentation-deprecated-sync 433
-Wdocumentation-html 433
-Wdocumentation-pedantic 433
-Wdocumentation-unknown-command 433
-Wdollar-in-identifier-extension 434
-Wdouble-promotion 434
-Wdtor-name 434
-Wdtor-typedef 434
-Wduplicate-decl-specifier 434
-Wduplicate-enum 434
-Wduplicate-method-arg 434
-Wduplicate-method-match 435
-Wduplicate-protocol 435
-Wdynamic-class-memaccess 435
-Wdynamic-exception-spec 435
-Weffc++ 435
-Welaborated-enum-base 435
-Welaborated-enum-class 435
-Wembedded-directive 435
-Wempty-body 435
-Wempty-decomposition 436
-Wempty-init-stmt 436
-Wempty-margins 436

-Wempty-translation-unit 436
-Wencode-type 436
-Wendif-labels 436
-Wenum-compare 436
-Wenum-compare-conditional 437
-Wenum-compare-switch 437
-Wenum-conversion 437
-Wenum-enum-conversion 437
-Wenum-float-conversion 437
-Wenum-too-large 437
-Wexceptions 437
-Wexcess-initializers 438
-Wexit-time-destructors 438
-Wexpansion-to-defined 438
-Wexplicit-initialize-call 438
-Wexplicit-ownership-type 438
-Wexport-unnamed 438
-Wexport-using-directive 438
-Wextern-c-compat 439
-Wextern-initializer 439
-Wextra 439
-Wextra-qualification 439
-Wextra-semi 439
-Wextra-semi-stmt 439
-Wextra-tokens 439
-Wfinal-dtor-non-final-class 440
-Wfinal-macro 440
-Wfixed-enum-extension 440
-Wfixed-point-overflow 440
-Wflag-enum 440
-Wflexible-array-extensions 440
-Wfloat-conversion 440
-Wfloat-equal 440
-Wfloat-overflow-conversion 441
-Wfloat-zero-conversion 441
-Wfor-loop-analysis 441
-Wformat 441
-Wformat-extra-args 442
-Wformat-insufficient-args 442
-Wformat-invalid-specifier 442
-Wformat-non-iso 442
-Wformat-nonliteral 442
-Wformat-pedantic 443
-Wformat-security 443
-Wformat-type-confusion 443
-Wformat-y2k 443

-Wformat-zero-length 443
-Wformat=2 443
-Wfortify-source 443
-Wfour-char-constants 443
-Wframe-address 444
-Wframe-larger-than 444
-Wframe-larger-than= 444
-Wframework-include-private-from-public 444
-Wfree-nonheap-object 444
-Wfunction-def-in-objc-container 444
-Wfunction-multiversion 444
-Wfuse-ld-path 444
-Wfuture-attribute-extensions 445
-Wfuture-compat 445
-Wgcc-compat 445
-Wglobal-constructors 445
-Wglobal-isel 445
-Wgnu 445
-Wgnu-alignof-expression 446
-Wgnu-anonymous-struct 446
-Wgnu-array-member-paren-init 446
-Wgnu-auto-type 446
-Wgnu-binary-literal 446
-Wgnu-case-range 446
-Wgnu-complex-integer 446
-Wgnu-compound-literal-initializer 446
-Wgnu-conditional-omitted-operand 447
-Wgnu-designator 447
-Wgnu-empty-initializer 447
-Wgnu-empty-struct 447
-Wgnu-flexible-array-initializer 447
-Wgnu-flexible-array-union-member 447
-Wgnu-folding-constant 447
-Wgnu-imaginary-constant 447
-Wgnu-include-next 448
-Wgnu-inline-cpp-without-extern 448
-Wgnu-label-as-value 448
-Wgnu-line-marker 448
-Wgnu-null-pointer-arithmetic 448
-Wgnu-pointer-arith 448
-Wgnu-redeclared-enum 448
-Wgnu-statement-expression 448
-Wgnu-statement-expression-from-macro-expansion 449
-Wgnu-static-float-init 449
-Wgnu-string-literal-operator-template 449
-Wgnu-union-cast 449

-Wgnu-variable-sized-type-not-at-end 449
-Wgnu-zero-line-directive 449
-Wgnu-zero-variadic-macro-arguments 449
-Wgpu-maybe-wrong-side 449
-Wheader-guard 449
-Wheader-hygiene 450
-Whip-only 450
-Whlsl-extensions 450
-Widiomatic-parentheses 450
-Wignored-attributes 450
-Wignored-availability-without-sdk-settings 453
-Wignored-optimization-argument 453
-Wignored-pragma-intrinsic 453
-Wignored-pragma-optimize 453
-Wignored-pragmas 453
-Wignored-qualifiers 455
-Wignored-reference-qualifiers 456
-Wimplicit 456
-Wimplicit-atomic-properties 456
-Wimplicit-const-int-float-conversion 456
-Wimplicit-conversion-floating-point-to-bool 456
-Wimplicit-exception-spec-mismatch 456
-Wimplicit-fallthrough 456
-Wimplicit-fallthrough-per-function 456
-Wimplicit-fixed-point-conversion 456
-Wimplicit-float-conversion 457
-Wimplicit-function-declaration 457
-Wimplicit-int 457
-Wimplicit-int-conversion 457
-Wimplicit-int-float-conversion 457
-Wimplicit-retain-self 458
-Wimplicitly-unsigned-literal 458
-Wimport 458
-Wimport-preprocessor-directive-pedantic 458
-Winaccessible-base 458
-Winclude-next-absolute-path 458
-Winclude-next-outside-header 458
-Wincompatible-exception-spec 458
-Wincompatible-function-pointer-types 458
-Wincompatible-library-redeclaration 459
-Wincompatible-ms-struct 459
-Wincompatible-pointer-types 459
-Wincompatible-pointer-types-discards-qualifiers 459
-Wincompatible-property-type 459
-Wincompatible-sysroot 459
-Wincomplete-framework-module-declaration 459

-Wincomplete-implementation 460
-Wincomplete-module 460
-Wincomplete-setjmp-declaration 460
-Wincomplete-umbrella 460
-Winconsistent-dllimport 460
-Winconsistent-missing-destructor-override 460
-Winconsistent-missing-override 460
-Wincrement-bool 460
-Winfinite-recursion 461
-Winit-self 461
-Winitializer-overrides 461
-Winjected-class-name 461
-Winline 461
-Winline-asm 461
-Winline-namespace-reopened-noninline 461
-Winline-new-delete 461
-Winstantiation-after-specialization 462
-Wint-conversion 462
-Wint-conversions 462
-Wint-in-bool-context 462
-Wint-to-pointer-cast 462
-Wint-to-void-pointer-cast 462
-Winteger-overflow 462
-Winterrupt-service-routine 462
-Winvalid-command-line-argument 463
-Winvalid-constexpr 463
-Winvalid-iboutlet 463
-Winvalid-initializer-from-system-header 463
-Winvalid-ios-deployment-target 463
-Winvalid-no-builtin-names 463
-Winvalid-noreturn 464
-Winvalid-offsetof 464
-Winvalid-or-nonexistent-directory 464
-Winvalid-partial-specialization 464
-Winvalid-pch 464
-Winvalid-pp-token 464
-Winvalid-source-encoding 464
-Winvalid-token-paste 465
-Wjump-seh-finally 465
-Wkeyword-compat 465
-Wkeyword-macro 465
-Wknr-promoted-parameter 465
-Wlanguage-extension-token 465
-Wlarge-by-value-copy 465
-Wliblto 465
-Wlinker-warnings 466

-Wliteral-conversion 466
-Wliteral-range 466
-Wlocal-type-template-args 466
-Wlogical-not-parentheses 466
-Wlogical-op-parentheses 466
-Wlong-long 466
-Wloop-analysis 467
-Wmacro-redefined 467
-Wmain 467
-Wmain-return-type 467
-Wmalformed-warning-check 467
-Wmany-braces-around-scalar-init 467
-Wmax-tokens 467
-Wmax-unsigned-zero 468
-Wmemset-transposed-args 468
-Wmemsize-comparison 468
-Wmethod-signatures 468
-Wmicrosoft 468
-Wmicrosoft-abstract 468
-Wmicrosoft-anon-tag 469
-Wmicrosoft-cast 469
-Wmicrosoft-charize 469
-Wmicrosoft-comment-paste 469
-Wmicrosoft-const-init 469
-Wmicrosoft-cpp-macro 469
-Wmicrosoft-default-arg-redefinition 469
-Wmicrosoft-drectve-section 469
-Wmicrosoft-end-of-file 470
-Wmicrosoft-enum-forward-reference 470
-Wmicrosoft-enum-value 470
-Wmicrosoft-exception-spec 470
-Wmicrosoft-exists 470
-Wmicrosoft-explicit-constructor-call 470
-Wmicrosoft-extra-qualification 470
-Wmicrosoft-fixed-enum 471
-Wmicrosoft-flexible-array 471
-Wmicrosoft-goto 471
-Wmicrosoft-inaccessible-base 471
-Wmicrosoft-include 471
-Wmicrosoft-mutable-reference 471
-Wmicrosoft-pure-definition 471
-Wmicrosoft-redeclare-static 471
-Wmicrosoft-sealed 471
-Wmicrosoft-static-assert 472
-Wmicrosoft-template 472
-Wmicrosoft-template-shadow 472

-Wmicrosoft-union-member-reference 472
-Wmicrosoft-unqualified-friend 472
-Wmicrosoft-using-decl 473
-Wmicrosoft-void-pseudo-dtor 473
-Wmisexpect 473
-Wmisleading-indentation 473
-Wmismatched-new-delete 473
-Wmismatched-parameter-types 473
-Wmismatched-return-types 473
-Wmismatched-tags 473
-Wmissing-braces 474
-Wmissing-constinit 474
-Wmissing-declarations 474
-Wmissing-exception-spec 474
-Wmissing-field-initializers 474
-Wmissing-format-attribute 474
-Wmissing-include-dirs 474
-Wmissing-method-return-type 474
-Wmissing-noescape 474
-Wmissing-noreturn 475
-Wmissing-prototype-for-cc 475
-Wmissing-prototypes 475
-Wmissing-selector-name 475
-Wmissing-sysroot 475
-Wmissing-variable-declarations 475
-Wmisspelled-assumption 475
-Rmodule-build 475
-Wmodule-conflict 476
-Wmodule-file-config-mismatch 476
-Wmodule-file-extension 476
-Rmodule-import 476
-Wmodule-import-in-extern-c 476
-Rmodule-lock 476
-Wmodules-ambiguous-internal-linkage 476
-Wmodules-import-nested-redundant 476
-Wmost 477
-Wmove 477
-Wmsvc-include 477
-Wmsvc-not-found 477
-Wmultichar 477
-Wmultiple-move-vbase 477
-Wnarrowing 477
-Wnested-anon-types 477
-Wnested-externs 477
-Wnew-returns-null 477
-Wnewline-eof 478

-Wnoderef 478
-Wnoexcept-type 478
-Wnon-c-typedef-for-linkage 478
-Wnon-gcc 478
-Wnon-literal-null-conversion 478
-Wnon-modular-include-in-framework-module 478
-Wnon-modular-include-in-module 478
-Wnon-pod-varargs 479
-Wnon-power-of-two-alignment 479
-Wnon-virtual-dtor 479
-Wnonnull 479
-Wnonportable-cfstrings 479
-Wnonportable-include-path 479
-Wnonportable-system-include-path 479
-Wnonportable-vector-initialization 479
-Wnontrivial-memaccess 480
-Wnsconsumed-mismatch 480
-Wnsreturns-mismatch 480
-Wnull-arithmetic 480
-Wnull-character 480
-Wnull-conversion 480
-Wnull-dereference 480
-Wnull-pointer-arithmetic 481
-Wnull-pointer-subtraction 481
-Wnullability 481
-Wnullability-completeness 481
-Wnullability-completeness-on-arrays 481
-Wnullability-declspec 481
-Wnullability-extension 481
-Wnullability-inferred-on-nested-type 482
-Wnullable-to-nonnull-conversion 482
-Wobjc-autosynthesis-property-ivar-name-match 482
-Wobjc-bool-constant-conversion 482
-Wobjc-boxing 482
-Wobjc-circular-container 482
-Wobjc-cocoa-api 482
-Wobjc-designated-initializers 482
-Wobjc-dictionary-duplicate-keys 483
-Wobjc-flexible-array 483
-Wobjc-forward-class-redefinition 483
-Wobjc-interface-ivars 483
-Wobjc-literal-compare 483
-Wobjc-literal-conversion 483
-Wobjc-macro-redefinition 483
-Wobjc-messaging-id 484
-Wobjc-method-access 484

-Wobjc-missing-property-synthesis 484
-Wobjc-missing-super-calls 484
-Wobjc-multiple-method-names 484
-Wobjc-noncopy-retain-block-property 484
-Wobjc-nonunified-exceptions 484
-Wobjc-property-assign-on-object-type 485
-Wobjc-property-implementation 485
-Wobjc-property-implicit-mismatch 485
-Wobjc-property-matches-cocoa-ownership-rule 485
-Wobjc-property-no-attribute 485
-Wobjc-property-synthesis 485
-Wobjc-protocol-method-implementation 485
-Wobjc-protocol-property-synthesis 486
-Wobjc-protocol-qualifiers 486
-Wobjc-readonly-with-setter-property 486
-Wobjc-redundant-api-use 486
-Wobjc-redundant-literal-use 486
-Wobjc-root-class 486
-Wobjc-signed-char-bool 486
-Wobjc-signed-char-bool-implicit-float-conversion 486
-Wobjc-signed-char-bool-implicit-int-conversion 487
-Wobjc-string-compare 487
-Wobjc-string-concatenation 487
-Wobjc-unsafe-perform-selector 487
-Wodr 487
-Wold-style-cast 488
-Wold-style-definition 488
-Wopencl-unsupported-rgba 488
-Wopenmp 488
-Wopenmp-51-extensions 488
-Wopenmp-clauses 488
-Wopenmp-loop-form 489
-Wopenmp-mapping 489
-Wopenmp-target 489
-Woption-ignored 489
-Wordered-compare-function-pointers 490
-Wout-of-line-declaration 490
-Wout-of-scope-function 490
-Wover-aligned 490
-Woverflow 491
-Woverlength-strings 491
-Woverloaded-shift-op-parentheses 491
-Woverloaded-virtual 491
-Woverride-init 491
-Woverride-module 491
-Woverriding-method-mismatch 491

-Woverriding-t-option 491
-Wpacked 492
-Wpadded 492
-Wparentheses 492
-Wparentheses-equality 492
-Wpartial-availability 492
-Rpass 492
-Rpass-analysis 492
-Wpass-failed 492
-Rpass-missed 493
-Wpch-date-time 493
-Wpedantic 493
-Wpedantic-core-features 496
-Wpedantic-macros 496
-Wpessimizing-move 496
-Wpointer-arith 496
-Wpointer-bool-conversion 496
-Wpointer-compare 496
-Wpointer-integer-compare 496
-Wpointer-sign 497
-Wpointer-to-enum-cast 497
-Wpointer-to-int-cast 497
-Wpointer-type-mismatch 497
-Wpoison-system-directories 497
-Wpotentially-direct-selector 497
-Wpotentially-evaluated-expression 497
-Wpragma-clang-attribute 497
-Wpragma-once-outside-header 498
-Wpragma-pack 498
-Wpragma-pack-suspicious-include 498
-Wpragma-system-header-outside-header 498
-Wpragmas 498
-Wpre-c++14-compat 498
-Wpre-c++14-compat-pedantic 499
-Wpre-c++17-compat 499
-Wpre-c++17-compat-pedantic 499
-Wpre-c++20-compat 500
-Wpre-c++20-compat-pedantic 500
-Wpre-c++2b-compat 501
-Wpre-c++2b-compat-pedantic 501
-Wpre-c2x-compat 501
-Wpre-c2x-compat-pedantic 501
-Wpre-openmp-51-compat 501
-Wpredefined-identifier-outside-function 501
-Wprivate-extern 502
-Wprivate-header 502

-Wprivate-module 502
-Wprofile-instr-missing 502
-Wprofile-instr-out-of-date 502
-Wprofile-instr-unprofiled 502
-Wproperty-access-dot-syntax 502
-Wproperty-attribute-mismatch 503
-Wprotocol 503
-Wprotocol-property-synthesis-ambiguity 503
-Wpsabi 503
-Wquoted-include-in-framework-header 503
-Wrange-loop-analysis 503
-Wrange-loop-bind-reference 503
-Wrange-loop-construct 503
-Wreadonly-iboutlet-property 504
-Wreceiver-expr 504
-Wreceiver-forward-class 504
-Wredeclared-class-member 504
-Wredundant-consteval-if 504
-Wredundant-decls 504
-Wredundant-move 504
-Wredundant-parens 504
-Wregister 504
-Wreinterpret-base-class 505
-Rremark-backend-plugin 505
-Wreorder 505
-Wreorder-ctor 505
-Wreorder-init-list 505
-Wrequires-super-attribute 505
-Wreserved-id-macro 505
-Wreserved-identifier 505
-Wreserved-macro-identifier 505
-Wreserved-user-defined-literal 506
-Wrestrict-expansion 506
-Wretained-language-linkage 506
-Wreturn-stack-address 506
-Wreturn-std-move 506
-Wreturn-type 506
-Wreturn-type-c-linkage 507
-Wrewrite-not-bool 507
-Rround-trip-cc1-args 507
-Wrtti 507
-Rsanitize-address 507
-Rsearch-path-usage 507
-Wsection 507
-Wselector 508
-Wselector-type-mismatch 508

-Wself-assign 508
-Wself-assign-field 508
-Wself-assign-overloaded 508
-Wself-move 508
-Wsemicolon-before-method-body 508
-Wsentinel 508
-Wsequence-point 509
-Wserialized-diagnostics 509
-Wshadow 509
-Wshadow-all 509
-Wshadow-field 509
-Wshadow-field-in-constructor 509
-Wshadow-field-in-constructor-modified 509
-Wshadow-ivar 509
-Wshadow-uncaptured-local 510
-Wshift-count-negative 510
-Wshift-count-overflow 510
-Wshift-negative-value 510
-Wshift-op-parentheses 510
-Wshift-overflow 510
-Wshift-sign-overflow 510
-Wshorten-64-to-32 510
-Wsign-compare 510
-Wsign-conversion 511
-Wsign-promo 511
-Wsigned-enum-bitfield 511
-Wsigned-unsigned-wchar 511
-Wsizeof-array-argument 511
-Wsizeof-array-decay 511
-Wsizeof-array-div 511
-Wsizeof-pointer-div 511
-Wsizeof-pointer-memaccess 512
-Wslash-u-filename 512
-Wslh-asm-goto 512
-Wsometimes-uninitialized 512
-Wsource-mgr 512
-Wsource-uses-openmp 512
-Wspir-compat 513
-Wspirv-compat 513
-Wstack-exhausted 513
-Wstack-protector 513
-Wstatic-float-init 513
-Wstatic-in-inline 513
-Wstatic-inline-explicit-instantiation 513
-Wstatic-local-in-inline 513
-Wstatic-self-init 514

-Wstdlibcxx-not-found 514
-Wstrict-aliasing 514
-Wstrict-aliasing=0 514
-Wstrict-aliasing=1 514
-Wstrict-aliasing=2 514
-Wstrict-overflow 514
-Wstrict-overflow=0 514
-Wstrict-overflow=1 514
-Wstrict-overflow=2 514
-Wstrict-overflow=3 514
-Wstrict-overflow=4 514
-Wstrict-overflow=5 514
-Wstrict-potentially-direct-selector 515
-Wstrict-prototypes 515
-Wstrict-selector-match 515
-Wstring-compare 515
-Wstring-concatenation 515
-Wstring-conversion 515
-Wstring-plus-char 515
-Wstring-plus-int 515
-Wstrlcpy-strlcat-size 516
-Wstrncat-size 516
-Wsuggest-destructor-override 516
-Wsuggest-override 516
-Wsuper-class-method-mismatch 516
-Wsuspicious-bzero 516
-Wsuspicious-memaccess 516
-Wswift-name-attribute 516
-Wswitch 517
-Wswitch-bool 517
-Wswitch-default 517
-Wswitch-enum 517
-Wsync-fetch-and-nand-semantics-changed 517
-Wsynth 518
-Wtarget-clones-mixed-specifiers 518
-Wtautological-bitwise-compare 518
-Wtautological-compare 518
-Wtautological-constant-compare 518
-Wtautological-constant-in-range-compare 518
-Wtautological-constant-out-of-range-compare 518
-Wtautological-objc-bool-compare 518
-Wtautological-overlap-compare 519
-Wtautological-pointer-compare 519
-Wtautological-type-limit-compare 519
-Wtautological-undefined-compare 519
-Wtautological-unsigned-char-zero-compare 519

-Wtautological-unsigned-enum-zero-compare 519
-Wtautological-unsigned-zero-compare 519
-Wtautological-value-range-compare 519
-Wtcb-enforcement 519
-Wtentative-definition-incomplete-type 520
-Wthread-safety 520
-Wthread-safety-analysis 520
-Wthread-safety-attributes 520
-Wthread-safety-beta 521
-Wthread-safety-negative 521
-Wthread-safety-precise 521
-Wthread-safety-reference 521
-Wthread-safety-verbose 521
-Wtrigraphs 521
-Wtype-limits 522
-Wtype-safety 522
-Wtypedef-redefinition 522
-Wtypename-missing 522
-Wunable-to-open-stats-file 522
-Wunaligned-access 522
-Wunaligned-qualifier-implicit-cast 522
-Wunavailable-declarations 522
-Wundeclared-selector 523
-Wundef 523
-Wundef-prefix 523
-Wundefined-bool-conversion 523
-Wundefined-func-template 523
-Wundefined-inline 523
-Wundefined-internal 523
-Wundefined-internal-type 523
-Wundefined-reinterpret-cast 523
-Wundefined-var-template 524
-Wunderaligned-exception-object 524
-Wunevaluated-expression 524
-Wunguarded-availability 524
-Wunguarded-availability-new 524
-Wunicode 524
-Wunicode-homoglyph 525
-Wunicode-whitespace 525
-Wunicode-zero-width 525
-Wuninitialized 525
-Wuninitialized-const-reference 525
-Wunknown-argument 525
-Wunknown-assumption 526
-Wunknown-attributes 526
-Wunknown-cuda-version 526

-Wunknown-directives 526
-Wunknown-escape-sequence 526
-Wunknown-pragmas 526
-Wunknown-sanitizers 527
-Wunknown-warning-option 527
-Wunnamed-type-template-args 527
-Wunneeded-internal-declaration 527
-Wunneeded-member-function 528
-Wunqualified-std-cast-call 528
-Wunreachable-code 528
-Wunreachable-code-aggressive 528
-Wunreachable-code-break 528
-Wunreachable-code-fallthrough 528
-Wunreachable-code-generic-assoc 528
-Wunreachable-code-loop-increment 528
-Wunreachable-code-return 528
-Wunsequenced 529
-Wunsupported-abi 529
-Wunsupported-abs 529
-Wunsupported-availability-guard 529
-Wunsupported-cb 529
-Wunsupported-dll-base-class-template 529
-Wunsupported-floating-point-opt 529
-Wunsupported-friend 530
-Wunsupported-gpopt 530
-Wunsupported-nan 530
-Wunsupported-target-opt 530
-Wunsupported-visibility 530
-Wunusable-partial-specialization 530
-Wunused 530
-Wunused-argument 531
-Wunused-but-set-parameter 531
-Wunused-but-set-variable 531
-Wunused-command-line-argument 531
-Wunused-comparison 531
-Wunused-const-variable 531
-Wunused-exception-parameter 532
-Wunused-function 532
-Wunused-getter-return-value 532
-Wunused-label 532
-Wunused-lambda-capture 532
-Wunused-local-typedef 532
-Wunused-local-typedefs 532
-Wunused-macros 532
-Wunused-member-function 532
-Wunused-parameter 532

-Wunused-private-field 533
-Wunused-property-ivar 533
-Wunused-result 533
-Wunused-template 533
-Wunused-value 533
-Wunused-variable 533
-Wunused-volatile-lvalue 534
-Wused-but-marked-unused 534
-Wuser-defined-literals 534
-Wuser-defined-warnings 534
-Wvarargs 534
-Wvariadic-macros 534
-Wvec-elem-size 534
-Wvector-conversion 535
-Wvector-conversions 535
-Wvexing-parse 535
-Wvisibility 535
-Wvla 535
-Wvla-extension 535
-Wvoid-pointer-to-enum-cast 535
-Wvoid-pointer-to-int-cast 535
-Wvoid-ptr-dereference 536
-Wvolatile-register-var 536
-Wwasm-exception-spec 536
-Wweak-template-vtables 536
-Wweak-vtables 536
-Wwritable-strings 536
-Wwrite-strings 536
-Wxor-used-as-pow 536
-Wzero-as-null-pointer-constant 536
-Wzero-length-array 537
Cross-compilation using Clang 537
Introduction 537
Cross compilation issues 537
General Cross-Compilation Options in Clang 537
Target Triple 537
CPU, FPU, ABI 538
Toolchain Options 538
Target-Specific Libraries 539
Multilibs 539
Clang Static Analyzer 539
Available Checkers 539
Default Checkers 544
core 544
core.CallAndMessage (C, C++, ObjC) 544
core.DivideZero (C, C++, ObjC) 545

core.NonNullParamChecker (C, C++, ObjC) 545
core.NullDereference (C, C++, ObjC) 545
core.StackAddressEscape (C) 546
core.UndefinedBinaryOperatorResult (C) 546
core.VLASize (C) 547
core.uninitialized.ArraySubscript (C) 547
core.uninitialized.Assign (C) 547
core.uninitialized.Branch (C) 547
core.uninitialized.CapturedBlockVariable (C) 547
core.uninitialized.UndefReturn (C) 547
cplusplus 548
cplusplus.InnerPointer (C++) 548
cplusplus.NewDelete (C++) 548
cplusplus.NewDeleteLeaks (C++) 549
cplusplus.PlacementNewChecker (C++) 549
cplusplus.SelfAssignment (C++) 549
cplusplus.StringChecker (C++) 549
deadcode 549
deadcode.DeadStores (C) 549
nullability 550
nullability.NullPassedToNonnull (ObjC) 550
nullability.NullReturnedFromNonnull (ObjC) 550
nullability.NullableDereferenced (ObjC) 550
nullability.NullablePassedToNonnull (ObjC) 550
nullability.NullableReturnedFromNonnull (ObjC) 551
optin 551
optin.cplusplus.UninitializedObject (C++) 551
optin.cplusplus.VirtualCall (C++) 552
optin.mpi.MPI-Checker (C) 553
optin.osx.cocoa.localizability.EmptyLocalizationContextChecker (ObjC) 553
optin.osx.cocoa.localizability.NonLocalizedStringChecker (ObjC) 553
optin.performance.GCDAntipattern 553
optin.performance.Padding 554
optin.portability.UnixAPI 554
security 554
security.FloatLoopCounter (C) 554
security.insecureAPI.UncheckedReturn (C) 554
security.insecureAPI.bcmp (C) 554
security.insecureAPI.bcopy (C) 554
security.insecureAPI.bzero (C) 554
security.insecureAPI.getpw (C) 554
security.insecureAPI.gets (C) 554
security.insecureAPI.mkstemp (C) 555
security.insecureAPI.mktemp (C) 555
security.insecureAPI.rand (C) 555
security.insecureAPI.strcpy (C) 555

security.insecureAPI.vfork (C) 555
security.insecureAPI.DeprecatedOrUnsafeBufferHandling (C) 555
unix 555
unix.API (C) 556
unix.Malloc (C) 556
unix.MallocSizeof (C) 557
unix.MismatchedDeallocator (C, C++) 557
unix.Vfork (C) 558
unix.cstring.BadSizeArg (C) 558
unix.cstring.NullArg (C) 559
osx 559
osx.API (C) 559
osx.NumberObjectConversion (C, C++, ObjC) 559
osx.ObjCProperty (ObjC) 559
osx.SecKeychainAPI (C) 559
osx.cocoa.AtSync (ObjC) 560
osx.cocoa.AutoreleaseWrite 561
osx.cocoa.ClassRelease (ObjC) 561
osx.cocoa.Dealloc (ObjC) 561
osx.cocoa.IncompatibleMethodTypes (ObjC) 562
osx.cocoa.Loops 562
osx.cocoa.MissingSuperCall (ObjC) 562
osx.cocoa.NSAutoreleasePool (ObjC) 562
osx.cocoa.NSError (ObjC) 562
osx.cocoa.NilArg (ObjC) 563
osx.cocoa.NonNilReturnValue 563
osx.cocoa.ObjCGenerics (ObjC) 563
osx.cocoa.RetainCount (ObjC) 563
osx.cocoa.RunLoopAutoreleaseLeak 564
osx.cocoa.SelfInit (ObjC) 564
osx.cocoa.SuperDealloc (ObjC) 564
osx.cocoa.UnusedIvars (ObjC) 564
osx.cocoa.VariadicMethodTypes (ObjC) 565
osx.coreFoundation.CFError (C) 565
osx.coreFoundation.CFNumber (C) 565
osx.coreFoundation.CFRetainRelease (C) 565
osx.coreFoundation.containers.OutOfBounds (C) 565
osx.coreFoundation.containers.PointerSizedValues (C) 565
Fuchsia 566
fuchsia.HandleChecker 566
WebKit 566
webkit.RefCntblBaseVirtualDtor 566
webkit.NoUncountedMemberChecker 566
webkit.UncountedLambdaCapturesChecker 567
Experimental Checkers 567
alpha.clone 567

alpha.clone.CloneChecker (C, C++, ObjC) 567
alpha.core 567
alpha.core.BoolAssignment (ObjC) 567
alpha.core.C11Lock 568
alpha.core.CallAndMessageUnInitRefArg (C,C++, ObjC) 568
alpha.core.CastSize (C) 568
alpha.core.CastToStruct (C, C++) 568
alpha.core.Conversion (C, C++, ObjC) 568
alpha.core.DynamicTypeChecker (ObjC) 569
alpha.core.FixedAddr (C) 569
alpha.core.IdenticalExpr (C, C++) 569
alpha.core.PointerArithm (C) 569
alpha.core.PointerSub (C) 570
alpha.core.SizeofPtr (C) 570
alpha.core.StackAddressAsyncEscape (C) 570
alpha.core.TestAfterDivZero (C) 570
alpha.cplusplus 570
alpha.cplusplus.DeleteWithNonVirtualDtor (C++) 570
alpha.cplusplus.EnumCastOutOfRange (C++) 571
alpha.cplusplus.InvalidatedIterator (C++) 571
alpha.cplusplus.IteratorRange (C++) 571
alpha.cplusplus.MismatchedIterator (C++) 571
alpha.cplusplus.MisusedMovedObject (C++) 572
alpha.cplusplus.SmartPtr (C++) 572
alpha.deadcode 572
alpha.deadcode.UnreachableCode (C, C++) 572
alpha.fuchsia 572
alpha.fuchsia.Lock 572
alpha.llvm 573
alpha.llvm.Conventions 573
alpha.osx 573
alpha.osx.cocoa.DirectIvarAssignment (ObjC) 573
alpha.osx.cocoa.DirectIvarAssignmentForAnnotatedFunctions (ObjC) 573
alpha.osx.cocoa.InstanceVariableInvalidation (ObjC) 573
alpha.osx.cocoa.MissingInvalidationMethod (ObjC) 574
alpha.osx.cocoa.localizability.PluralMisuseChecker (ObjC) 574
alpha.security 575
alpha.security.ArrayBound (C) 575
alpha.security.ArrayBoundV2 (C) 575
alpha.security.MallocOverflow (C) 576
alpha.security.MmapWriteExec (C) 576
alpha.security.ReturnPtrRange (C) 576
alpha.security.cert 577
alpha.security.cert.pos 577
alpha.security.cert.pos.34c 577
alpha.security.cert.env 577

alpha.security.cert.env.InvalidPtr 577
alpha.security.taint 578
alpha.security.taint.TaintPropagation (C, C++) 578
alpha.unix 579
alpha.unix.StdCLibraryFunctionArgs (C) 579
alpha.unix.BlockInCriticalSection (C) 580
alpha.unix.Chroot (C) 580
alpha.unix.PthreadLock (C) 580
alpha.unix.SimpleStream (C) 581
alpha.unix.Stream (C) 581
alpha.unix.cstring.BufferOverlap (C) 582
alpha.unix.cstring.NotNullTerminated (C) 582
alpha.unix.cstring.OutOfBounds (C) 582
alpha.unix.cstring.UninitializedRead (C) 582
alpha.nondeterminism.PointerIteration (C++) 583
alpha.nondeterminism.PointerSorting (C++) 583
alpha.WebKit 583
alpha.webkit.UncountedCallArgsChecker 583
alpha.webkit.UncountedLocalVarsChecker 584
Debug Checkers 585
debug 585
debug.AnalysisOrder 585
debug.ConfigDumper 585
debug.DumpCFG Display 585
debug.DumpCallGraph 585
debug.DumpCalls 586
debug.DumpDominators 586
debug.DumpLiveVars 586
debug.DumpTraversal 586
debug.ExprInspection 586
debug.Stats 586
debug.TaintTest 586
debug.ViewCFG 586
debug.ViewCallGraph 586
debug.ViewExplodedGraph 586
User Docs 586
Cross Translation Unit (CTU) Analysis 586
Overview 587
PCH-based analysis 587
Manual CTU Analysis 587
Automated CTU Analysis with CodeChecker 588
Automated CTU Analysis with scan-build-py (don’t do it) 589
On-demand analysis 589
Manual CTU Analysis 589
Automated CTU Analysis with CodeChecker 591
Automated CTU Analysis with scan-build-py (don’t do it) 591

Taint Analysis Configuration 592
Overview 592
Example configuration file 593
Configuration file syntax and semantics 594
Filter syntax and semantics 594
Propagation syntax and semantics 594
Sink syntax and semantics 595
Developer Docs 595
Debug Checks 595
General Analysis Dumpers 595
Path Tracking 596
State Checking 596
ExprInspection checks 596
Statistics 599
Output testing checkers 599
Inlining 599
c++-inlining 600
c++-template-inlining 600
c++-stdlib-inlining 600
c++-container-inlining 600
Basics of Implementation 601
Retry Without Inlining 601
Deciding When to Inline 601
Dynamic Calls and Devirtualization 602
DynamicTypeInfo 602
RuntimeDefinition 602
Inlining Dynamic Calls 602
Bifurcation 603
Objective-C Message Heuristics 603
C++ Caveats 603
CallEvent 604
Initializer List 604
Nullability Checks 608
Inlining 609
Annotations on multi level pointers 609
Implementation notes 609
Region Store 610
Binding Invalidation 610
ObjCIvarRegions 610
Region Invalidation 611
Default Bindings 611
Lazy Bindings (LazyCompoundVal) 611
Thread Safety Analysis 612
Introduction 612
Getting Started 612
Running The Analysis 613

Basic Concepts: Capabilities 613
Reference Guide 613
GUARDED_BY(c) and PT_GUARDED_BY(c) 614
REQUIRES(…), REQUIRES_SHARED(…) 614
ACQUIRE(…), ACQUIRE_SHARED(…), RELEASE(…), RELEASE_SHARED(…), RELEASE_GENERIC(…) 614
EXCLUDES(…) 615
NO_THREAD_SAFETY_ANALYSIS 616
RETURN_CAPABILITY(c) 616
ACQUIRED_BEFORE(…), ACQUIRED_AFTER(…) 616
CAPABILITY(<string>) 617
SCOPED_CAPABILITY 617
TRY_ACQUIRE(<bool>, …), TRY_ACQUIRE_SHARED(<bool>, …) 617
ASSERT_CAPABILITY(…) and ASSERT_SHARED_CAPABILITY(…) 617
GUARDED_VAR and PT_GUARDED_VAR 617
Warning flags 617
Negative Capabilities 618
Frequently Asked Questions 619
Known Limitations 619
Lexical scope 619
Private Mutexes 619
No conditionally held locks. 620
No checking inside constructors and destructors. 620
No inlining. 620
No alias analysis. 621
ACQUIRED_BEFORE(…) and ACQUIRED_AFTER(…) are currently unimplemented. 621
mutex.h 621
Data flow analysis: an informal introduction 626
Abstract 626
Data flow analysis 626
The purpose of data flow analysis 626
Sample problem and an ad-hoc solution 626
Too much information and “top” values 627
Uninitialized variables and “bottom” values 628
A practical lattice that tracks sets of concrete values 628
Formalization 628
Symbolic execution: a very short informal introduction 629
Symbolic values 629
Flow condition 630
Symbolic pointers 630
Example: finding output parameters 630
Problem description 631
Lattice design 632
Using data flow results to identify output parameters 632
Example: finding dead stores 633
Example: definitive initialization 633
Example: refactoring raw pointers to `unique_ptr` 634

Example: finding redundant branch conditions 636
Example: finding unchecked `std::optional` unwraps 636
Example: finding dead code behind A/B experiment flags 637
Example: finding inefficient usages of associative containers 637
Example: refactoring types that implicitly convert to each other 638
AddressSanitizer 639
Introduction 639
How to build 639
Usage 640
Symbolizing the Reports 640
Additional Checks 641
Initialization order checking 641
Stack Use After Return (UAR) 641
Memory leak detection 641
Issue Suppression 641
Suppressing Reports in External Libraries 641
Conditional Compilation with `__has_feature(address_sanitizer)` 642
Disabling Instrumentation with `__attribute__((no_sanitize("address")))` 642
Suppressing Errors in Recompiled Code (Ignorelist) 642
Suppressing memory leaks 642
Code generation control 643
Instrumentation code outlining 643
Limitations 643
Supported Platforms 643
Current Status 643
More Information 643
ThreadSanitizer 643
Introduction 643
How to build 644
Supported Platforms 644
Usage 644
`__has_feature(thread_sanitizer)` 644
`__attribute__((no_sanitize("thread")))` 645
`__attribute__((disable_sanitizer_instrumentation))` 645
Ignorelist 645
Limitations 645
Current Status 645
More Information 645
MemorySanitizer 646
Introduction 646
How to build 646
Usage 646
`__has_feature(memory_sanitizer)` 647
`__attribute__((no_sanitize("memory")))` 647
`__attribute__((disable_sanitizer_instrumentation))` 647
Ignorelist 647

- Report symbolization 647
- Origin Tracking 647
- Use-after-destruction detection 648
- Handling external code 648
- Supported Platforms 648
- Limitations 648
- Current Status 649
- More Information 649

UndefinedBehaviorSanitizer 649

- Introduction 649
- How to build 650
- Usage 650
- Available checks 650
  - Volatile 652
- Minimal Runtime 652
- Stack traces and report symbolization 652
- Logging 652
- Silencing Unsigned Integer Overflow 653
- Issue Suppression 653
  - Disabling Instrumentation with `__attribute__((no_sanitize("undefined")))` 653
  - Suppressing Errors in Recompiled Code (Ignorelist) 653
  - Runtime suppressions 653
- Supported Platforms 653
- Current Status 654
- Additional Configuration 654
  - Example 654
- More Information 654

DataFlowSanitizer 654

- DataFlowSanitizer Design Document 654
  - Use Cases 654
  - Interface 654
  - Taint label representation 656
  - Origin tracking trace representation 656
  - Memory layout and label management 656
  - Propagating labels through arguments 657
  - Implementing the ABI list 657
  - Checking ABI Consistency 658
- Introduction 658
- How to build libc++ with DFSan 658
- Usage 659
  - ABI List 659
  - Compilation Flags 660
  - Environment Variables 661
- Example 661
- Origin Tracking 662
- Current status 663

Design 663
LeakSanitizer 663
Introduction 663
Usage 663
Supported Platforms 663
More Information 664
SanitizerCoverage 664
Introduction 664
Tracing PCs with guards 664
Inline 8bit-counters 666
Inline bool-flag 666
PC-Table 666
Tracing PCs 667
Instrumentation points 667
Edge coverage 667
Tracing data flow 667
Disabling instrumentation with `__attribute__((no_sanitize("coverage")))` 668
Disabling instrumentation without source modification 669
Default implementation 669
Sancov data format 670
Sancov Tool 670
Coverage Reports 670
Output directory 670
SanitizerStats 671
Introduction 671
How to build and run 671
Sanitizer special case list 671
Introduction 672
Goal and usage 672
Example 672
Format 672
Control Flow Integrity 673
Control Flow Integrity Design Documentation 673
Forward-Edge CFI for Virtual Calls 673
Optimizations 674
Stripping Leading/Trailing Zeros in Bit Vectors 674
Short Inline Bit Vectors 675
Virtual Table Layout 675
Alignment 676
Padding to Powers of 2 676
Eliminating Bit Vector Checks for All-Ones Bit Vectors 677
Forward-Edge CFI for Virtual Calls by Interleaving Virtual Tables 677
Split virtual table groups into separate virtual tables 677
Order virtual tables by a pre-order traversal of the class hierarchy 677
Interleave virtual tables 678
Forward-Edge CFI for Indirect Function Calls 679

Shared library support 680
CallSiteTypeId 681
CFI_Check 681
CFI Shadow 681
CFI_SlowPath 682
Position-independent executable requirement 682
Backward-edge CFI for return statements (RCFI) 682
Leaf Functions 682
Functions called once 682
Functions called in a small number of call sites 682
General case 683
Returns from functions called indirectly 683
Cross-DSO calls 683
Non-goals 683
Hardware support 683
Introduction 684
Available schemes 685
Trapping and Diagnostics 685
Forward-Edge CFI for Virtual Calls 685
Performance 685
Bad Cast Checking 685
Non-Virtual Member Function Call Checking 686
Strictness 686
Indirect Function Call Checking 686
`-fsanitize-cfi-icall-generalize-pointers` 686
`-fsanitize-cfi-canonical-jump-tables` 687
`-fsanitize=cfi-icall` and `-fsanitize=function` 687
Member Function Pointer Call Checking 688
Ignorelist 688
Shared library support 688
Design 688
Publications 688
LTO Visibility 689
Example 689
SafeStack 690
Introduction 690
Performance 691
Compatibility 691
Known compatibility limitations 691
Security 691
Known security limitations 691
Usage 692
Supported Platforms 692
Low-level API 692
`__has_feature(safe_stack)` 692
`__attribute__((no_sanitize("safe-stack")))` 692

__builtin___get_unsafe_stack_ptr() 692
__builtin___get_unsafe_stack_bottom() 692
__builtin___get_unsafe_stack_top() 692
__builtin___get_unsafe_stack_start() 692
Design 692
setjmp and exception handling 693
Publications 693
ShadowCallStack 693
Introduction 693
Comparison 693
Compatibility 693
Security 694
Usage 694
Low-level API 695
__has_feature(shadow_call_stack) 695
__attribute__((no_sanitize("shadow-call-stack"))) 695
Example 695
Source-based Code Coverage 696
Introduction 696
The code coverage workflow 696
Compiling with coverage enabled 697
Running the instrumented program 697
Creating coverage reports 697
Exporting coverage data 699
Interpreting reports 699
Format compatibility guarantees 699
Impact of llvm optimizations on coverage reports 700
Using the profiling runtime without static initializers 700
Using the profiling runtime without a filesystem 700
Collecting coverage reports for the llvm project 700
Drawbacks and limitations 700
Clang implementation details 701
Gap regions 701
Branch regions 701
Switch statements 701
Modules 702
Introduction 702
Problems with the current model 703
Semantic import 703
Problems modules do not solve 704
Using Modules 704
Objective-C Import declaration 704
Includes as imports 704
Module maps 705
Compilation model 705
Command-line parameters 705

-cc1 Options 707
Using Prebuilt Modules 707
Module Semantics 708
Macros 709
Module Map Language 709
Lexical structure 710
Module map file 710
Module declaration 710
Requires declaration 711
Header declaration 713
Umbrella directory declaration 714
Submodule declaration 714
Export declaration 715
Re-export Declaration 716
Use declaration 716
Link declaration 717
Configuration macros declaration 717
Conflict declarations 718
Attributes 718
Private Module Map Files 718
Modularizing a Platform 719
Future Directions 720
Where To Learn More About Modules 720
MSVC compatibility 721
ABI features 721
Template instantiation and name lookup 722
Misexpect 722
OpenCL Support 724
Missing features or with limited support 724
Internals Manual 724
OpenCL Metadata 724
OpenCL Specific Options 724
OpenCL builtins 725
OpenCL Extensions and Features 725
Implementation guidelines 726
Address spaces attribute 726
C++ for OpenCL Implementation Status 727
Missing features or with limited support 727
OpenCL C 3.0 Usage 727
OpenCL C 3.0 Implementation Status 727
Experimental features 728
C++ libraries for OpenCL 728
OpenMP Support 729
General improvements 729
Cuda devices support 729
Directives execution modes 729

- Data-sharing modes 730
- Features not supported or with limited support for Cuda devices 730

OpenMP 5.0 Implementation Details 730

OpenMP 5.1 Implementation Details 733

OpenMP Extensions 735

SYCL Compiler and Runtime architecture design 735

- Introduction 735
- Address space handling 735

HLSL Support 737

- Introduction 737
- Project Goals 737
  - Non-Goals 737
- Guiding Principles 737
- Architectural Direction 737
  - DXC Driver 738
  - Parser 738
  - Sema 738
  - CodeGen 738
- HLSL Language 738
  - An Aside on GPU Languages 738
  - Pointers & References 739
  - HLSL `this` Keyword 739
  - Bitshifts 739
  - Non-short Circuiting Logical Operators 739
  - Precise Qualifier 739
  - Differences in Templates 739
  - Vector Extensions 739
  - Standard Library 739
  - Unsupported C & C++ Features 739

ThinLTO 740

- Introduction 740
- Current Status 741
  - Clang/LLVM 741
  - Linkers 741
- Usage 741
  - Basic 741
  - Controlling Backend Parallelism 741
  - Incremental 742
  - Cache Pruning 742
  - Clang Bootstrap 742
- More Information 743

API Notes: Annotations Without Modifying Headers 743

- Usage 743
- Limitations 744
- “Versioned” API Notes 744
- Reference 744

Clang “man” pages 747
Basic Commands 747
clang - the Clang C, C++, and Objective-C compiler 747
SYNOPSIS 747
DESCRIPTION 747
OPTIONS 748
Stage Selection Options 748
Language Selection and Mode Options 748
Target Selection Options 751
Code Generation Options 751
Driver Options 753
Diagnostics Options 754
Preprocessor Options 754
ENVIRONMENT 754
BUGS 755
SEE ALSO 755
diagtool - clang diagnostics tool 755
SYNOPSIS 755
DESCRIPTION 755
SUBCOMMANDS 755
find-diagnostic-id 755
list-warnings 755
show-enabled 755
tree 755
Frequently Asked Questions (FAQ) 756
Driver 756
I run `clang -cc1 ...` and get weird errors about missing headers 756
I get errors about some headers being missing (`stddef.h`, `stdarg.h`) 756
Using Clang as a Library **756**
Choosing the Right Interface for Your Application 756
LibClang 756
Clang Plugins 757
LibTooling 757
External Clang Examples 758
Introduction 758
List of projects and tools 758
Introduction to the Clang AST 759
Introduction 759
Examining the AST 759
AST Context 760
AST Nodes 760
LibTooling 760
Introduction 760
Parsing a code snippet in memory 760
Writing a standalone tool 761
Parsing common tools options 761

Creating and running a ClangTool 761
Putting it together — the first tool 761
Running the tool on some code 762
Builtin includes 763
Linking 763
LibFormat 763
Design 763
Interface 763
Style Options 763
Clang Plugins 764
Introduction 764
Writing a `PluginASTAction` 764
Registering a plugin 764
Defining pragmas 764
Defining attributes 765
Putting it all together 765
Running the plugin 765
Using the compiler driver 765
Using the cc1 command line 766
Using the clang command line 766
Interaction with `-clear-ast-before-backend` 766
How to write RecursiveASTVisitor based ASTFrontendActions. 766
Introduction 766
Creating a FrontendAction 766
Creating an ASTConsumer 767
Using the RecursiveASTVisitor 767
Accessing the SourceManager and ASTContext 768
Putting it all together 768
Tutorial for building tools using LibTooling and LibASTMatchers 769
Step 0: Obtaining Clang 769
Step 1: Create a ClangTool 770
Intermezzo: Learn AST matcher basics 772
Step 2: Using AST matchers 772
Step 3.5: More Complicated Matchers 773
Step 4: Retrieving Matched Nodes 775
Matching the Clang AST 776
Introduction 776
How to create a matcher 777
Binding nodes in match expressions 777
Writing your own matchers 777
`VariadicDynCastAllOfMatcher<Base, Derived>` 777
`AST_MATCHER_P(Type, Name, ParamType, Param)` 777
Matcher creation functions 777
Clang Transformer Tutorial 777
What is Clang Transformer? 778
Who is Clang Transformer for? 778

Getting Started 778
Example: style-checking names 779
Example: renaming a function 779
Example: method to function 779
Example: rewriting method calls 780
Reference: ranges, stencils, edits, rules 780
Rewriting ASTs to… Text? 780
Range Selectors 780
Stencils 781
Edits 781
`EditGenerator`s (Advanced) 781
Rules 782
Using a RewriteRule as a clang-tidy check 782
Related Reading 782
ASTImporter: Merging Clang ASTs 782
Introduction 783
Algorithm of the import 783
API 784
Errors during the import process 787
Error propagation 788
Polluted AST 788
Using the `-ast-merge` Clang front-end action 790
Example for C 790
Example for C++ 791
How To Setup Clang Tooling For LLVM 791
Introduction 791
Setup Clang Tooling Using CMake and Make 791
Setup Clang Tooling Using CMake on Windows 792
Using Clang Tools 792
Using Ninja Build System 793
JSON Compilation Database Format Specification 794
Background 794
Supported Systems 794
Format 794
Build System Integration 795
Alternatives 795
Clang's refactoring engine 795
Introduction 796
Refactoring Action Rules 796
Rule Types 796
How to Create a Rule 797
Refactoring Action Rule Requirements 797
Selection Requirements 798
Other Requirements 798
Refactoring Options 798
Using Clang Tools **798**

Overview 798
Clang Tools Organization 799
Core Clang Tools 799
`clang-check` 799
`clang-format` 799
Extra Clang Tools 799
`clang-tidy` 799
Ideas for new Tools 799
ClangCheck 800
ClangFormat 800
Standalone Tool 801
Vim Integration 802
Emacs Integration 802
BBEdit Integration 803
CLion Integration 803
Visual Studio Integration 803
Visual Studio Code Integration 803
Script for patch reformatting 803
Current State of Clang Format for LLVM 804
Clang-Format Style Options 804
Configuring Style with clang-format 804
Disabling Formatting on a Piece of Code 805
Configuring Style in Code 805
Configurable Format Style Options 805
Adding additional style options 860
Examples 860
Clang Formatted Status 861
Clang Linker Wrapper 914
Introduction 915
Usage 915
Example 915
Clang Nvlink Wrapper 915
Introduction 915
Use Case 916
Working 916
Clang Offload Bundler 916
Introduction 916
Supported File Formats 917
Bundled Text File Layout 917
Bundled Binary File Layout 917
Bundle Entry ID 918
Target ID 919
Target Specific information 919
Archive Unbundling 920
Clang Offload Wrapper 920
Introduction 920

Usage 920
Example 921
OpenMP Device Binary Embedding 921
Enum Types 921
Structure Types 921
Global Variables 922
Binary Descriptor for Device Images 922
Global Constructor and Destructor 923
Image Binary Embedding and Execution for OpenMP 923
Clang Offload Packager 923
Introduction 923
Binary Format 924
Usage 925
Example 925
Design Documents **926**
"Clang" CFE Internals Manual 926
Introduction 927
LLVM Support Library 928
The Clang "Basic" Library 928
The Diagnostics Subsystem 928
The `Diagnostic*Kinds.td` files 928
The Format String 929
Formatting a Diagnostic Argument 929
Producing the Diagnostic 932
Fix-It Hints 932
The `DiagnosticConsumer` Interface 933
Adding Translations to Clang 933
The `SourceLocation` and `SourceManager` classes 933
`SourceRange` and `CharSourceRange` 934
The Driver Library 934
Precompiled Headers 934
The Frontend Library 934
Compiler Invocation 934
Command Line Interface 934
Command Line Parsing 935
Command Line Generation 935
Adding new Command Line Option 935
Option Marshalling Infrastructure 937
Option Marshalling Annotations 938
The Lexer and Preprocessor Library 939
The Token class 940
Annotation Tokens 940
The `Lexer` class 941
The `TokenLexer` class 942
The `MultipleIncludeOpt` class 942
The Parser Library 942

The AST Library 942
Design philosophy 942
Immutability 942
Faithfulness 943
The `Type` class and its subclasses 943
Canonical Types 944
The `QualType` class 944
Declaration names 945
Declaration contexts 946
Redeclarations and Overloads 946
Lexical and Semantic Contexts 947
Transparent Declaration Contexts 947
Multiply-Defined Declaration Contexts 948
Error Handling 949
Recovery AST 949
Types and dependence 950
ContainsErrors bit 950
The ASTImporter 951
Abstract Syntax Graph 951
Structural Equivalency 951
Redeclaration Chains 952
Traversal during the Import 953
Error Handling 953
Lookup Problems 954
ExternalASTSource 955
Class Template Instantiations 955
Visibility of Declarations 956
Strategies to Handle Conflicting Names 956
The `CFG` class 956
Basic Blocks 956
Entry and Exit Blocks 956
Conditional Control-Flow 957
Constant Folding in the Clang AST 958
Implementation Approach 958
Extensions 959
The Sema Library 959
The CodeGen Library 959
How to change Clang 959
How to add an attribute 959
Attribute Basics 960
`include/clang/Basic/Attr.td` 960
Spellings 960
Subjects 961
Documentation 961
Arguments 962
Other Properties 962

Boilerplate 963
Semantic handling 963
How to add an expression or statement 963
Driver Design & Internals 965
Introduction 966
Features and Goals 966
GCC Compatibility 966
Flexible 966
Low Overhead 966
Simple 966
Internal Design and Implementation 966
Internals Introduction 967
Design Overview 967
Driver Stages 968
Additional Notes 971
The Compilation Object 971
Unified Parsing & Pipelining 971
ToolChain Argument Translation 971
Unused Argument Warnings 971
Relation to GCC Driver Concepts 971
Offloading Design & Internals 972
Introduction 972
OpenMP Offloading 972
Offloading Overview 972
Compilation Process 973
Generating Offloading Entries 973
Accessing Entries on the Device 974
Debugging Information 974
Offload Device Compilation 974
Creating Fat Objects 975
Linking Target Device Code 975
Device Binary Wrapping 975
Structure Types 975
Global Variables 976
Binary Descriptor for Device Images 976
Global Constructor and Destructor 977
Offloading Example 977
Precompiled Header and Modules Internals 978
Using Precompiled Headers with `clang` 978
Design Philosophy 978
AST File Contents 979
Metadata Block 980
Source Manager Block 981
Preprocessor Block 981
Types Block 981
Declarations Block 981

Statements and Expressions 982
Identifier Table Block 982
Method Pool Block 983
AST Reader Integration Points 983
Chained precompiled headers 984
Modules 984
ABI tags 985
Introduction 985
Declaration 985
Mangling 985
Active tags 985
Required tags for a function 986
Required tags for a variable 986
Available tags 986
Hardware-assisted AddressSanitizer Design Documentation 986
Introduction 986
Algorithm 986
Short granules 987
Instrumentation 987
Memory Accesses 987
Heap 988
Stack 988
Globals 988
Error reporting 989
Attribute 989
Comparison with AddressSanitizer 989
Supported architectures 989
Related Work 990
Constant Interpreter 990
Introduction 990
Bytecode Compilation 990
Primitive Types 990
Composite types 991
Bytecode Execution 991
Memory Organisation 991
Blocks 991
Descriptors 992
Pointers 993
BlockPointer 993
ExternPointer 994
TargetPointer 994
TypeInfoPointer 994
InvalidPointer 994
TODO 994
Missing Language Features 994
Known Bugs 995

Clang pre-release 15 Release Notes

Introduction 2
What's New in Clang pre-release 15? 2
Potentially Breaking Changes 2
Major New Features 2
Bug Fixes 2
Non-comprehensive list of changes in this release 5
New Compiler Flags 5
Deprecated Compiler Flags 5
Modified Compiler Flags 5
Removed Compiler Flags 5
New Pragmas in Clang 5
Attribute Changes in Clang 6
Windows Support 6
AIX Support 6
C Language Changes in Clang 6
C2x Feature Support 6
C++ Language Changes in Clang 7
CUDA/HIP Language Changes in Clang 7
Objective-C Language Changes in Clang 8
OpenCL C Language Changes in Clang 8
ABI Changes in Clang 8
OpenMP Support in Clang 8
CUDA Support in Clang 8
X86 Support in Clang 8
DWARF Support in Clang 8
Arm and AArch64 Support in Clang 8
Floating Point Support in Clang 8
Internal API Changes 8
Build System Changes 8
AST Matchers 8
clang-format 8
libclang 9
Static Analyzer 9
Undefined Behavior Sanitizer (UBSan) 9
Core Analysis Improvements 9
New Issues Found 9
Python Binding Changes 9
Significant Known Problems 9
Additional Information 9

Written by the LLVM Team with modifications by IBM.

Introduction

This document contains the release notes for the Clang C/C++/Objective-C frontend, part of the LLVM Compiler Infrastructure, pre-release 15, on which Open XL C/C++ 17.1.1 is based. Open XL C/C++ 17.1.1 provides a customized subset of the LLVM Compiler infrastructure. This documentation is provided as-is and for reference only. It does not constitute support of all features in the Open XL C/C++ 17.1.1 product. Refer to the Open XL C/C++ compiler documentation for features that are officially supported. Here we describe the status of Clang in some detail, including major improvements from the previous LLVM release and new feature work. For the general LLVM release notes, see the LLVM documentation. All LLVM releases may be downloaded from the LLVM releases web site.

For more information about Clang or LLVM, including information about the latest release, please see the Clang Web Site or the LLVM Web Site.

Note that if you are reading this file from a Git checkout or the main Clang web page, this document applies to the *next* release, not the current one. To see the release notes for a specific release, please see the releases page.

What's New in Clang pre-release 15?

Some of the major new features and improvements to Clang are listed here. Generic improvements to Clang as a whole or to its underlying infrastructure are described first, followed by language-specific sections with improvements to Clang's support for those languages.

Potentially Breaking Changes

These changes are ones which we think may surprise users when upgrading to Clang 15 because of the opportunity they pose for disruption to existing code bases.

- The `-Wimplicit-function-declaration` and `-Wimplicit-int` warning diagnostics are now enabled by default in C99, C11, and C17. As of C2x, support for implicit function declarations and implicit int has been removed, and the warning options will have no effect. Specifying `-Wimplicit-int` in C89 mode will now issue warnings instead of being a noop. *NOTE* these warnings are expected to default to an error in Clang 16. We recommend that projects using configure scripts verify the results do not change before/after setting `-Werror=implicit-function-declarations` or `-Wimplicit-int` to avoid incompatibility with Clang 16.

Major New Features

- Clang now supports the `-fzero-call-used-regs` feature for x86. The purpose of this feature is to limit Return-Oriented Programming (ROP) exploits and information leakage. It works by zeroing out a selected class of registers before function return — e.g., all GPRs that are used within the function. There is an analogous `zero_call_used_regs` attribute to allow for finer control of this feature.
- Clang now supports randomizing structure layout in C. This feature is a compile-time hardening technique, making it more difficult for an attacker to retrieve data from structures. Specify randomization with the `randomize_layout` attribute. The corresponding `no_randomize_layout` attribute can be used to turn the feature off.

  A seed value is required to enable randomization, and is deterministic based on a seed value. Use the `-frandomize-layout-seed=` or `-frandomize-layout-seed-file=` flags.

  > **Note**
  >
  > Randomizing structure layout is a C-only feature.

Bug Fixes

- `CXXNewExpr::getArraySize()` previously returned a `llvm::Optional` wrapping a `nullptr` when the `CXXNewExpr` did not have an array size expression. This was fixed and `::getArraySize()` will now always either return `None` or a `llvm::Optional` wrapping a valid `Expr*`. This fixes Issue 53742.
- We now ignore full expressions when traversing cast subexpressions. This fixes Issue 53044.
- Allow `-Wno-gnu` to silence GNU extension diagnostics for pointer arithmetic diagnostics. Fixes Issue 54444.
- Placeholder constraints, as in `Concept auto x = f();`, were not checked when modifiers like `auto&` or `auto**` were added. These constraints are now checked. This fixes Issue 53911 and Issue 54443.
- Previously invalid member variables with template parameters would crash clang. Now fixed by setting identifiers for them. This fixes Issue 28475 (PR28101).
- Now allow the `restrict` and `_Atomic` qualifiers to be used in conjunction with `__auto_type` to match the behavior in GCC. This fixes Issue 53652.
- No longer crash when specifying a variably-modified parameter type in a function with the `naked` attribute. This fixes Issue 50541.
- Allow multiple `#pragma weak` directives to name the same undeclared (if an alias, target) identifier instead of only processing one such `#pragma weak` per identifier. Fixes Issue 28985.
- Assignment expressions in C11 and later mode now properly strip the _Atomic qualifier when determining the type of the assignment expression. Fixes Issue 48742.
- Improved the diagnostic when accessing a member of an atomic structure or union object in C; was previously an unhelpful error, but now issues a `-Watomic-access` warning which defaults to an error. Fixes Issue 54563.
- Unevaluated lambdas in dependant contexts no longer result in clang crashing. This fixes Issues 50376, 51414, 51416, and 51641.
- The builtin function __builtin_dump_struct would crash clang when the target struct contains a bitfield. It now correctly handles bitfields. This fixes Issue Issue 54462.
- Statement expressions are now disabled in default arguments in general. This fixes Issue Issue 53488.
- According to CWG 1394 and C++20 [dcl.fct.def.general]p2, Clang should not diagnose incomplete types in function definitions if the function body is “= delete;”. This fixes Issue Issue 52802.
- Unknown type attributes with a `[[]]` spelling are no longer diagnosed twice. This fixes Issue Issue 54817.
- Clang should no longer incorrectly diagnose a variable declaration inside of a lambda expression that shares the name of a variable in a containing if/while/for/switch init statement as a redeclaration. This fixes Issue 54913.
- Overload resolution for constrained function templates could use the partial order of constraints to select an overload, even if the parameter types of the functions were different. It now diagnoses this case correctly as an ambiguous call and an error. Fixes Issue 53640.
- No longer crash when trying to determine whether the controlling expression argument to a generic selection expression has side effects in the case where the expression is result dependent. This fixes Issue 50227.
- Fixed an assertion when constant evaluating an initializer for a GCC/Clang floating-point vector type when the width of the initialization is exactly the same as the elements of the vector being initialized. Fixes Issue 50216.
- Fixed a crash when the `__bf16` type is used such that its size or alignment is calculated on a target which does not support that type. This fixes Issue 50171.
- Fixed a false positive diagnostic about an unevaluated expression having no side effects when the expression is of VLA type and is an operand of the `sizeof` operator. Fixes Issue 48010.
- Fixed a false positive diagnostic about scoped enumerations being a C++11 extension in C mode. A scoped enumeration’s enumerators cannot be named in C because there is no way to fully qualify the enumerator name, so this “extension” was unintentional and useless. This fixes Issue 42372.
- Clang will now find and emit a call to an allocation function in a promise_type body for coroutines if there is any allocation function declaration in the scope of promise_type. Additionally, to implement CWG2585, a coroutine will no longer generate a call to a global allocation function with the signature (std::size_t, p0, …, pn). This fixes Issue Issue 54881.

- Implement CWG 2394: Const class members may be initialized with a defaulted default constructor under the same conditions it would be allowed for a const object elsewhere.
- `__has_unique_object_representations` no longer reports that `_BitInt` types have unique object representations if they have padding bits.

Improvements to Clang's diagnostics

- `-Wliteral-range` will warn on floating-point equality comparisons with constants that are not representable in a casted value. For example, `(float) f == 0.1` is always false.
- `-Winline-namespace-reopened-noninline` now takes into account that the `inline` keyword must appear on the original but not necessarily all extension definitions of an inline namespace and therefore points its note at the original definition. This fixes Issue 50794 (PR51452).
- `-Wunused-but-set-variable` now also warns if the variable is only used by unary operators.
- `-Wunused-variable` no longer warn for references extending the lifetime of temporaries with side effects. This fixes Issue 54489.
- Modified the behavior of `-Wstrict-prototypes` and added a new, related diagnostic `-Wdeprecated-non-prototype`. The strict prototypes warning will now only diagnose deprecated declarations and definitions of functions without a prototype where the behavior in C2x will remain correct. This diagnostic remains off by default but is now enabled via `-pedantic` due to it being a deprecation warning. `-Wstrict-prototypes` has no effect in C2x or when `-fno-knr-functions` is enabled. `-Wdeprecated-non-prototype` will diagnose cases where the deprecated declarations or definitions of a function without a prototype will change behavior in C2x. Additionally, it will diagnose calls which pass arguments to a function without a prototype. This warning is enabled only when the `-Wdeprecated-non-prototype` option is enabled at the function declaration site, which allows a developer to disable the diagnostic for all callers at the point of declaration. This diagnostic is grouped under the `-Wstrict-prototypes` warning group, but is enabled by default. `-Wdeprecated-non-prototype` has no effect in C2x or when `-fno-knr-functions` is enabled.
- Clang now appropriately issues an error in C when a definition of a function without a prototype and with no arguments is an invalid redeclaration of a function with a prototype. e.g., `void f(int); void f() {}` is now properly diagnosed.
- No longer issue a "declaration specifiers missing, defaulting to int" diagnostic in C89 mode because it is not an extension in C89, it was valid code. The diagnostic has been removed entirely as it did not have a diagnostic group to disable it, but it can be covered wholly by `-Wimplicit-int`.
- `-Wmisexpect` warns when the branch weights collected during profiling conflict with those added by `llvm.expect`.
- `-Wthread-safety-analysis` now considers overloaded compound assignment and increment/decrement operators as writing to their first argument, thus requiring an exclusive lock if the argument is guarded.
- `-Wenum-conversion` now warns on converting a signed enum of one type to an unsigned enum of a different type (or vice versa) rather than `-Wsign-conversion`.
- Added the `-Wunreachable-code-generic-assoc` diagnostic flag (grouped under the `-Wunreachable-code` flag) which is enabled by default and warns the user about `_Generic` selection associations which are unreachable because the type specified is an array type or a qualified type.
- Added the `-Wgnu-line-marker` diagnostic flag (grouped under the `-Wgnu` flag) which is a portability warning about use of GNU linemarker preprocessor directives. Fixes Issue 55067.
- Using `#elifdef` and `#elifndef` that are incompatible with C/C++ standards before C2x/C++2b are now warned via `-pedantic`. Additionally, on such language mode, `-Wpre-c2x-compat` and `-Wpre-c++2b-compat` diagnostic flags report a compatibility issue. Fixes Issue 55306.
- Clang now checks for stack resource exhaustion when recursively parsing declarators in order to give a diagnostic before we run out of stack space. This fixes Issue 51642.
- Unknown preprocessor directives in a skipped conditional block are now given a typo correction suggestion if the given directive is sufficiently similar to another preprocessor conditional directive. For example, if `#esle`

appears in a skipped block, we will warn about the unknown directive and suggest `#else` as an alternative. `#elifdef` and `#elifndef` are only suggested when in C2x or C++2b mode. Fixes Issue 51598.

- The `-Wdeprecated` diagnostic will now warn on out-of-line `constexpr` declarations downgraded to definitions in C++1z, in addition to the existing warning on out-of-line `const` declarations.

Non-comprehensive list of changes in this release

- Improve __builtin_dump_struct:
  - Support bitfields in struct and union.
  - Improve the dump format, dump both bitwidth(if its a bitfield) and field value.
  - Remove anonymous tag locations and flatten anonymous struct members.
  - Beautify dump format, add indent for struct members.
  - Support passing additional arguments to the formatting function, allowing use with `fprintf` and similar formatting functions.
  - Support use within constant evaluation in C++, if a `constexpr` formatting function is provided.
  - Support formatting of base classes in C++.
  - Support calling a formatting function template in C++, which can provide custom formatting for non-aggregate types.
- Previously disabled sanitizer options now enabled by default: - ASAN_OPTIONS=detect_stack_use_after_return=1 (only on Linux). - MSAN_OPTIONS=poison_in_dtor=1.

New Compiler Flags

- Added the `-fno-knr-functions` flag to allow users to opt into the C2x behavior where a function with an empty parameter list is treated as though the parameter list were `void`. There is no `-fknr-functions` or `-fno-no-knr-functions` flag; this feature cannot be disabled in language modes where it is required, such as C++ or C2x.
- A new ARM pass to workaround Cortex-A57 Erratum 1742098 and Cortex-A72 Erratum 1655431 can be enabled using `-mfix-cortex-a57-aes-1742098` or `-mfix-cortex-a72-aes-1655431`. The pass is enabled when using either of these cpus with `-mcpu=` and can be disabled using `-mno-fix-cortex-a57-aes-1742098` or `-mno-fix-cortex-a72-aes-1655431`.

Deprecated Compiler Flags

Modified Compiler Flags

Removed Compiler Flags

- Removed the `-fno-concept-satisfaction-caching` flag. The flag was added at the time when the draft of C++20 standard did not permit caching of atomic constraints. The final standard permits such caching, see WG21 P2104R0.

New Pragmas in Clang

- Added support for MSVC's `#pragma function`, which tells the compiler to generate calls to functions listed in the pragma instead of using the builtins.
- Added support for MSVC's `#pragma alloc_text`. The pragma names the code section functions are placed in. The pragma only applies to functions with C linkage.
- …

Attribute Changes in Clang

- Added support for parameter pack expansion in `clang::annotate`.
- The `overloadable` attribute can now be written in all of the syntactic locations a declaration attribute may appear. This fixes Issue 53805.
- Improved namespace attributes handling:
  - Handle GNU attributes before a namespace identifier and subsequent attributes of different kinds.
  - Emit error on GNU attributes for a nested namespace definition.
- Statement attributes `[[clang::noinline]]` and `[[clang::always_inline]]` can be used to control inlining decisions at callsites.
- `#pragma clang attribute push` now supports multiple attributes within a single directive.
- The `__declspec(naked)` attribute can no longer be written on a member function in Microsoft compatibility mode, matching the behavior of cl.exe.
- Attribute `no_builtin` should now affect the generated code. It now disables builtins (corresponding to the specific names listed in the attribute) in the body of the function the attribute is on.

Windows Support

- Add support for MSVC-compatible `/JMC`/`/JMC-` flag in clang-cl (supports X86/X64/ARM/ARM64). `/JMC` could only be used when `/Zi` or `/Z7` is turned on. With this addition, clang-cl can be used in Visual Studio for the JustMyCode feature. Note, you may need to manually add `/JMC` as additional compile options in the Visual Studio since it currently assumes clang-cl does not support `/JMC`.

AIX Support

- The driver no longer adds `-mignore-xcoff-visibility` by default for AIX targets when no other visibility command-line options are in effect, as ignoring hidden visibility can silently have undesirable side effects (e.g when libraries depend on visibility to hide non-ABI facing entities). The `-mignore-xcoff-visibility` option can be manually specified on the command-line to recover the previous behavior if desired.

C Language Changes in Clang

- Finished implementing support for DR423. We already correctly handled stripping qualifiers from cast expressions, but we did not strip qualifiers on function return types. We now properly treat the function as though it were declared with an unqualified, non-atomic return type. Fixes Issue 39595.

C2x Feature Support

- Implemented WG14 N2674 The noreturn attribute.
- Implemented WG14 N2935 Make false and true first-class language features.
- Implemented WG14 N2763 Adding a fundamental type for N-bit integers.
- Implemented WG14 N2775 Literal suffixes for bit-precise integers.
- Implemented the `*_WIDTH` macros to complete support for WG14 N2412 Two's complement sign representation for C2x.
- Implemented WG14 N2418 Adding the u8 character prefix.
- Removed support for implicit function declarations. This was a C89 feature that was removed in C99, but cannot be supported in C2x because it requires support for functions without prototypes, which no longer exist in C2x.
- Implemented WG14 N2841 No function declarators without prototypes and WG14 N2432 Remove support for function definitions with identifier lists.

C++ Language Changes in Clang

- Improved `-O0` code generation for calls to `std::move`, `std::forward`, `std::move_if_noexcept`, `std::addressof`, and `std::as_const`. These are now treated as compiler builtins and implemented directly, rather than instantiating the definition from the standard library.
- Fixed mangling of nested dependent names such as `T::a::b`, where `T` is a template parameter, to conform to the Itanium C++ ABI and be compatible with GCC. This breaks binary compatibility with code compiled with earlier versions of clang; use the `-fclang-abi-compat=14` option to get the old mangling.
- Preprocessor character literals with a `u8` prefix are now correctly treated as unsigned character literals. This fixes Issue 54886.
- Stopped allowing constraints on non-template functions to be compliant with dcl.decl.general p4.

C++20 Feature Support

- Diagnose consteval and constexpr issues that happen at namespace scope. This partially addresses Issue 51593.
- No longer attempt to evaluate a consteval UDL function call at runtime when it is called through a template instantiation. This fixes Issue 54578.
- Implemented `__builtin_source_location()`, which enables library support for `std::source_location`.
- The mangling scheme for C++20 modules has incompatibly changed. The initial mangling was discovered not to be reversible, and the weak ownership design decision did not give the backwards compatibility that was hoped for. C++20 since added `extern "C++"` semantics that can be used for such compatibility. The demangler now demangles symbols with named module attachment.
- Enhanced the support for C++20 Modules, including: Partitions, Reachability, Header Unit and `extern "C++"` semantics.
- Implemented P1103R3: Merging Modules.
- Implemented P1779R3: ABI isolation for member functions.
- Implemented P1874R1: Dynamic Initialization Order of Non-Local Variables in Modules.
- Partially implemented P1815R2: Translation-unit-local entities.
- As per “Conditionally Trivial Special Member Functions” (P0848), it is now possible to overload destructors using concepts. Note that the rest of the paper about other special member functions is not yet implemented.
- Skip rebuilding lambda expressions in arguments of immediate invocations. This fixes GH56183, GH51695, GH50455, GH54872, GH54587.

C++2b Feature Support

- Implemented P2128R6: Multidimensional subscript operator.
- Implemented P0849R8: auto(x): decay-copy in the language.
- Implemented P2242R3: Non-literal variables (and labels and gotos) in constexpr functions.

CUDA/HIP Language Changes in Clang

- Added *__noinline__* as a keyword to avoid diagnostics due to usage of *__attribute__((__noinline__))* in CUDA/HIP programs.

Objective-C Language Changes in Clang

OpenCL C Language Changes in Clang

…

ABI Changes in Clang

- GCC doesn't pack non-POD members in packed structs unless the packed attribute is also specified on the member. Clang historically did perform such packing. Clang now matches the gcc behavior (except on Darwin and PS4). You can switch back to the old ABI behavior with the flag: `-fclang-abi-compat=14.0`.

OpenMP Support in Clang

…

CUDA Support in Clang

- …

X86 Support in Clang

- Support `-mharden-sls=[none|all|return|indirect-jmp]` for straight-line speculation hardening.

DWARF Support in Clang

Arm and AArch64 Support in Clang

Floating Point Support in Clang

Internal API Changes

- Added a new attribute flag `AcceptsExprPack` that when set allows expression pack expansions in the parsed arguments of the corresponding attribute. Additionally it introduces delaying of attribute arguments, adding common handling for creating attributes that cannot be fully initialized prior to template instantiation.

Build System Changes

- CMake `-DCLANG_DEFAULT_PIE_ON_LINUX=ON` is now the default. This is used by linux-gnu systems to decide whether `-fPIE -pie` is the default (instead of `-fno-pic -no-pie`). This matches GCC installations on many Linux distros. Note: linux-android and linux-musl always default to `-fPIE -pie`, ignoring this variable. `-DCLANG_DEFAULT_PIE_ON_LINUX` may be removed in the future.

AST Matchers

- Expanded `isInline` narrowing matcher to support C++17 inline variables.
- Added `forEachTemplateArgument` matcher which creates a match every time a `templateArgument` matches the matcher supplied to it.

clang-format

- **Important change**: Renamed `IndentRequires` to `IndentRequiresClause` and changed the default for all styles from `false` to `true`.
- Reworked and improved handling of concepts and requires. Added the `RequiresClausePosition` option as part of that.
- Changed `BreakBeforeConceptDeclarations` from `Boolean` to an enum.
- Option `InsertBraces` has been added to insert optional braces after control statements.

libclang

- …

Static Analyzer

- New CTU implementation that keeps the slow-down around 2x compared to the single-TU analysis, even in case of complex C++ projects. Still, it finds the majority of the “old” CTU findings. Besides, not more than ~3% of the bug reports are lost compared to single-TU analysis, the lost reports are highly likely to be false positives.
- Added a new checker `alpha.unix.cstring.UninitializedRead` this will check for uninitialized reads from common memory copy/manipulation functions such as `memcpy`, `mempcpy`, `memmove`, `memcmp`, `` `*strcmp*` ``, `strncmp`, `strcpy`, `strlen`, `strsep` and many more. Although this checker currently is in list of alpha checkers due to a false positive.

Undefined Behavior Sanitizer (UBSan)

Core Analysis Improvements

- …

New Issues Found

- …

Python Binding Changes

The following methods have been added:

- …

Significant Known Problems

Additional Information

A wide variety of additional information is available on the Clang web page. The web page contains versions of the API documentation which are up-to-date with the Git version of the source code. You can access versions of these documents specific to this release by going into the “`clang/docs/`” directory in the Clang tree.

Using Clang as a Compiler

Clang Compiler User's Manual

Introduction **12**

- Terminology 12
- Basic Usage 12

Command Line Options **13**

- Options to Control Error and Warning Messages 13
  - Formatting of Diagnostics 13
  - Individual Warning Groups 17
- Options to Control Clang Crash Diagnostics 18
- Options to Emit Optimization Reports 18
  - Current limitations 19
- Options to Emit Resource Consumption Reports 19
- Other Options 20
- Configuration files 20

Language and Target-Independent Features **21**

- Controlling Errors and Warnings 21
  - Controlling How Clang Displays Diagnostics 21
  - Diagnostic Mappings 22
  - Diagnostic Categories 22
  - Controlling Diagnostics via Command Line Flags 22
  - Controlling Diagnostics via Pragmas 22
  - Controlling Diagnostics in System Headers 23
  - Controlling Deprecation Diagnostics in Clang-Provided C Runtime Headers 23
  - Enabling All Diagnostics 24
  - Controlling Static Analyzer Diagnostics 24
- Precompiled Headers 24
  - Generating a PCH File 24
  - Using a PCH File 24
  - Relocatable PCH Files 25
- Controlling Floating Point Behavior 25
  - A note about `__FLT_EVAL_METHOD__` 29
  - A note about Floating Point Constant Evaluation 29
- Controlling Code Generation 30
- Profile Guided Optimization 33
  - Differences Between Sampling and Instrumentation 34
  - Using Sampling Profilers 34
    - Sample Profile Formats 34
    - Sample Profile Text Format 35
  - Profiling with Instrumentation 36
  - Disabling Instrumentation 38
  - Instrumenting only selected files or functions 38
  - Profile remapping 39

GCOV-based Profiling 40
Controlling Debug Information 40
Controlling Size of Debug Information 40
Controlling Macro Debug Info Generation 41
Controlling Debugger “Tuning” 41
Controlling LLVM IR Output 41
Controlling Value Names in LLVM IR 41
Comment Parsing Options 42
C Language Features 42
Extensions supported by clang 42
Differences between various standard modes 42
GCC extensions not implemented yet 43
Intentionally unsupported GCC extensions 43
Microsoft extensions 44
C++ Language Features 44
Controlling implementation limits 44
Objective-C Language Features 44
Objective-C++ Language Features 44
OpenMP Features 44
Controlling implementation limits 45
OpenCL Features 45
OpenCL Specific Options 45
OpenCL Targets 46
Specific Targets 46
Generic Targets 46
OpenCL Header 47
OpenCL Extensions 47
OpenCL-Specific Attributes 47
nosvm 47
opencl_unroll_hint 48
convergent 48
noduplicate 48
C++ for OpenCL 49
Constructing and destroying global objects 49
Libraries 50
Target-Specific Features and Limitations 50
CPU Architectures Features and Limitations 50
X86 50
ARM 50
PowerPC 50
Other platforms 50
Operating System Features and Limitations 51
Windows 51
Cygwin 51
MinGW32 51

MinGW-w64 51
AIX 51
SPIR-V support 52
clang-cl 52
Command-Line Options 53
The /clang: Option 57
The /Zc:dllexportInlines- Option 57
Finding Clang runtime libraries 58

Introduction

The Clang Compiler is an open-source compiler for the C family of programming languages, aiming to be the best in class implementation of these languages. Clang builds on the LLVM optimizer and code generator, allowing it to provide high-quality optimization and code generation support for many targets. For more general information, please see the Clang Web Site or the LLVM Web Site.

This document describes important notes about using Clang as a compiler for an end-user, documenting the supported features, command line options, etc. If you are interested in using Clang to build a tool that processes code, please see “Clang” CFE Internals Manual. If you are interested in the Clang Static Analyzer, please see its web page.

Clang is one component in a complete toolchain for C family languages. A separate document describes the other pieces necessary to assemble a complete toolchain.

Clang is designed to support the C family of programming languages, which includes C, Objective-C, C++, and Objective-C++ as well as many dialects of those. For language-specific information, please see the corresponding language specific section:

- C Language: K&R C, ANSI C89, ISO C90, ISO C94 (C89+AMD1), ISO C99 (+TC1, TC2, TC3).
- Objective-C Language: ObjC 1, ObjC 2, ObjC 2.1, plus variants depending on base language.
- C++ Language
- Objective C++ Language
- OpenCL Kernel Language: OpenCL C 1.0, 1.1, 1.2, 2.0, 3.0, and C++ for OpenCL 1.0 and 2021.

In addition to these base languages and their dialects, Clang supports a broad variety of language extensions, which are documented in the corresponding language section. These extensions are provided to be compatible with the GCC, Microsoft, and other popular compilers as well as to improve functionality through Clang-specific features. The Clang driver and language features are intentionally designed to be as compatible with the GNU GCC compiler as reasonably possible, easing migration from GCC to Clang. In most cases, code “just works”. Clang also provides an alternative driver, clang-cl, that is designed to be compatible with the Visual C++ compiler, cl.exe.

In addition to language specific features, Clang has a variety of features that depend on what CPU architecture or operating system is being compiled for. Please see the Target-Specific Features and Limitations section for more details.

The rest of the introduction introduces some basic compiler terminology that is used throughout this manual and contains a basic introduction to using Clang as a command line compiler.

Terminology

Front end, parser, backend, preprocessor, undefined behavior, diagnostic, optimizer

Basic Usage

Intro to how to use a C compiler for newbies.

compile + link compile then link debug info enabling optimizations picking a language to use, defaults to C17 by default. Autosenses based on extension. using a makefile

Command Line Options

This section is generally an index into other sections. It does not go into depth on the ones that are covered by other sections. However, the first part introduces the language selection and other high level options like **-c**, **-g**, etc.

Options to Control Error and Warning Messages

-Werror
Turn warnings into errors.

-Werror=foo

Turn warning “foo” into an error.

-Wno-error=foo
Turn warning “foo” into a warning even if **-Werror** is specified.

-Wfoo
Enable warning “foo”. See the diagnostics reference for a complete list of the warning flags that can be specified in this way.

-Wno-foo
Disable warning “foo”.

-w
Disable all diagnostics.

-Weverything
Enable all diagnostics.

-pedantic
Warn on language extensions.

-pedantic-errors
Error on language extensions.

-Wsystem-headers
Enable warnings from system headers.

-ferror-limit=123
Stop emitting diagnostics after 123 errors have been produced. The default is 20, and the error limit can be disabled with *-ferror-limit=0*.

-ftemplate-backtrace-limit=123
Only emit up to 123 template instantiation notes within the template instantiation backtrace for a single warning or error. The default is 10, and the limit can be disabled with *-ftemplate-backtrace-limit=0*.

Formatting of Diagnostics

Clang aims to produce beautiful diagnostics by default, particularly for new users that first come to Clang. However, different people have different preferences, and sometimes Clang is driven not by a human, but by a program that wants consistent and easily parsable output. For these cases, Clang provides a wide range of options to control the exact output format of the diagnostics that it generates.

-f[no-]show-column

Print column number in diagnostic.

This option, which defaults to on, controls whether or not Clang prints the column number of a diagnostic. For example, when this is enabled, Clang will print something like:

```
test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]
#endif bad
       ^
       //
```

When this is disabled, Clang will print “test.c:28: warning…” with no column number.

The printed column numbers count bytes from the beginning of the line; take care if your source contains multibyte characters.

-f[no-]show-source-location

Print source file/line/column information in diagnostic.

This option, which defaults to on, controls whether or not Clang prints the filename, line number and column number of a diagnostic. For example, when this is enabled, Clang will print something like:

```
test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]
#endif bad
       ^
       //
```

When this is disabled, Clang will not print the “test.c:28:8: ” part.

-f[no-]caret-diagnostics

Print source line and ranges from source code in diagnostic. This option, which defaults to on, controls whether or not Clang prints the source line, source ranges, and caret when emitting a diagnostic. For example, when this is enabled, Clang will print something like:

```
test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]
#endif bad
       ^
       //
```

-f[no-]color-diagnostics

This option, which defaults to on when a color-capable terminal is detected, controls whether or not Clang prints diagnostics in color.

When this option is enabled, Clang will use colors to highlight specific parts of the diagnostic, e.g.,

When this is disabled, Clang will just print:

```
test.c:2:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]
#endif bad
       ^
       //
```

-fansi-escape-codes

Controls whether ANSI escape codes are used instead of the Windows Console API to output colored diagnostics. This option is only used on Windows and defaults to off.

-fdiagnostics-format=clang/msvc/vi

Changes diagnostic output format to better match IDEs and command line tools.
This option controls the output format of the filename, line number, and column printed in diagnostic messages. The options, and their affect on formatting a simple conversion diagnostic, follow:

clang (default)

```
t.c:3:11: warning: conversion specifies type 'char *' but the argument has type 'int'
```

msvc

```
t.c(3,11) : warning: conversion specifies type 'char *' but the argument has type 'int'
```

vi

```
t.c +3:11: warning: conversion specifies type 'char *' but the argument has type 'int'
```

-f[no-]diagnostics-show-option

Enable `[-Woption]` information in diagnostic line.

This option, which defaults to on, controls whether or not Clang prints the associated warning group option name when outputting a warning diagnostic. For example, in this output:

```
test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]
#endif bad
       ^
       //
```

Passing **-fno-diagnostics-show-option** will prevent Clang from printing the [-Wextra-tokens] information in the diagnostic. This information tells you the flag needed to enable or disable the diagnostic, either from the command line or through #pragma GCC diagnostic.

-fdiagnostics-show-category=none/id/name

Enable printing category information in diagnostic line.

This option, which defaults to “none”, controls whether or not Clang prints the category associated with a diagnostic when emitting it. Each diagnostic may or many not have an associated category, if it has one, it is listed in the diagnostic categorization field of the diagnostic line (in the []’s).

For example, a format string warning will produce these three renditions based on the setting of this option:

```
t.c:3:11: warning: conversion specifies type 'char *' but the argument has type 'int' [-Wformat]
t.c:3:11: warning: conversion specifies type 'char *' but the argument has type 'int' [-Wformat,1]
t.c:3:11: warning: conversion specifies type 'char *' but the argument has type 'int' [-Wformat,Format String]
```

This category can be used by clients that want to group diagnostics by category, so it should be a high level category. We want dozens of these, not hundreds or thousands of them.

-f[no-]save-optimization-record[=<format>]

Enable optimization remarks during compilation and write them to a separate file.

This option, which defaults to off, controls whether Clang writes optimization reports to a separate file. By recording diagnostics in a file, users can parse or sort the remarks in a convenient way.

By default, the serialization format is YAML.

The supported serialization formats are:

- `-fsave-optimization-record=yaml`: A structured YAML format.
- `-fsave-optimization-record=bitstream`: A binary format based on LLVM Bitstream.

The output file is controlled by -foptimization-record-file.

In the absence of an explicit output file, the file is chosen using the following scheme:

`<base>.opt.<format>`

where `<base>` is based on the output file of the compilation (whether it’s explicitly specified through *-o* or not) when used with *-c* or *-S*. For example:

- `clang -fsave-optimization-record -c in.c -o out.o` will generate `out.opt.yaml`
- `clang -fsave-optimization-record -c in.c `` will generate ``in.opt.yaml`

When targeting (Thin)LTO, the base is derived from the output filename, and the extension is not dropped.

When targeting ThinLTO, the following scheme is used:

`<base>.opt.<format>.thin.<num>.<format>`

Darwin-only: when used for generating a linked binary from a source file (through an intermediate object file), the driver will invoke *cc1* to generate a temporary object file. The temporary remark file will be emitted next to the object file, which will then be picked up by *dsymutil* and emitted in the .dSYM bundle. This is available for all formats except YAML.

For example:

`clang -fsave-optimization-record=bitstream in.c -o out` will generate

- `/var/folders/43/9y164hh52tv_2nrdxrj31nyw0000gn/T/a-9be59b.o`
- `/var/folders/43/9y164hh52tv_2nrdxrj31nyw0000gn/T/a-9be59b.opt.bitstream`
- `out`
- `out.dSYM/Contents/Resources/Remarks/out`

Darwin-only: compiling for multiple architectures will use the following scheme:

`<base>-<arch>.opt.<format>`

Note that this is incompatible with passing the -foptimization-record-file option.

-foptimization-record-file

Control the file to which optimization reports are written. This implies -fsave-optimization-record.

On Darwin platforms, this is incompatible with passing multiple `-arch <arch>` options.

-foptimization-record-passes

Only include passes which match a specified regular expression.

When optimization reports are being output (see -fsave-optimization-record), this option controls the passes that will be included in the final report.

If this option is not used, all the passes are included in the optimization record.

-f[no-]diagnostics-show-hotness

Enable profile hotness information in diagnostic line.

This option controls whether Clang prints the profile hotness associated with diagnostics in the presence of profile-guided optimization information. This is currently supported with optimization remarks (see Options to Emit Optimization Reports). The hotness information allows users to focus on the hot optimization remarks that are likely to be more relevant for run-time performance.

For example, in this output, the block containing the callsite of *foo* was executed 3000 times according to the profile data:

```
s.c:7:10: remark: foo inlined into bar (hotness: 3000) [-Rpass-analysis=inline]
  sum += foo(x, x - 2);
         ^
```

This option is implied when -fsave-optimization-record is used. Otherwise, it defaults to off.

-fdiagnostics-hotness-threshold

Prevent optimization remarks from being output if they do not have at least this hotness value.

This option, which defaults to zero, controls the minimum hotness an optimization remark would need in order to be output by Clang. This is currently supported with optimization remarks (see Options to Emit Optimization Reports) when profile hotness information in diagnostics is enabled (see -fdiagnostics-show-hotness).

-f[no-]diagnostics-fixit-info

Enable “FixIt” information in the diagnostics output.

This option, which defaults to on, controls whether or not Clang prints the information on how to fix a specific diagnostic underneath it when it knows. For example, in this output:

```
test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]
#endif bad
       ^
       //
```

Passing **-fno-diagnostics-fixit-info** will prevent Clang from printing the “//” line at the end of the message. This information is useful for users who may not understand what is wrong, but can be confusing for machine parsing.

-fdiagnostics-print-source-range-info

Print machine parsable information about source ranges. This option makes Clang print information about source ranges in a machine parsable format after the file/line/column number information. The information is a simple sequence of brace enclosed ranges, where each range lists the start and end line/column locations. For example, in this output:

```
exprs.c:47:15:{47:8-47:14}{47:17-47:24}: error: invalid operands to binary expression ('int *' and '_Complex float')
   P = (P-42) + Gamma*4;
       ~~~~~~ ^ ~~~~~~~
```

The {}’s are generated by -fdiagnostics-print-source-range-info.

The printed column numbers count bytes from the beginning of the line; take care if your source contains multibyte characters.

-fdiagnostics-parseable-fixits

Print Fix-Its in a machine parseable form.

This option makes Clang print available Fix-Its in a machine parseable format at the end of diagnostics. The following example illustrates the format:

```
fix-it:"t.cpp":{7:25-7:29}:"Gamma"
```

The range printed is a half-open range, so in this example the characters at column 25 up to but not including column 29 on line 7 in t.cpp should be replaced with the string “Gamma”. Either the range or the replacement string may be empty (representing strict insertions and strict erasures, respectively). Both the file name and the insertion string escape backslash (as “\”), tabs (as “\t”), newlines (as “\n”), double quotes(as “\””) and non-printable characters (as octal “\xxx”).

The printed column numbers count bytes from the beginning of the line; take care if your source contains multibyte characters.

-fno-elide-type
Turns off elision in template type printing.
The default for template type printing is to elide as many template arguments as possible, removing those which are the same in both template types, leaving only the differences. Adding this flag will print all the template arguments. If supported by the terminal, highlighting will still appear on differing arguments.
Default:

```
t.cc:4:5: note: candidate function not viable: no known conversion from 'vector<map<[...], map<float, [...]>>>' to 'vector<map<[...], map<double, [...]>>>' for 1st argument;
```

-fno-elide-type:

```
t.cc:4:5: note: candidate function not viable: no known conversion from 'vector<map<int, map<float, int>>>' to 'vector<map<int, map<double, int>>>' for 1st argument;
```

-fdiagnostics-show-template-tree
Template type diffing prints a text tree.
For diffing large templated types, this option will cause Clang to display the templates as an indented text tree, one argument per line, with differences marked inline. This is compatible with -fno-elide-type.
Default:

```
t.cc:4:5: note: candidate function not viable: no known conversion from 'vector<map<[...], map<float, [...]>>>' to 'vector<map<[...], map<double, [...]>>>' for 1st argument;
```

With **-fdiagnostics-show-template-tree**:

```
t.cc:4:5: note: candidate function not viable: no known conversion for 1st argument;
  vector<
    map<
      [...],
      map<
        [float != double],
        [...]>>>
```

Individual Warning Groups

TODO: Generate this from tblgen. Define one anchor per warning group.

-Wextra-tokens
Warn about excess tokens at the end of a preprocessor directive.
This option, which defaults to on, enables warnings about extra tokens at the end of preprocessor directives. For example:

```
test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]
#endif bad
       ^
```

These extra tokens are not strictly conforming, and are usually best handled by commenting them out.

-Wambiguous-member-template
Warn about unqualified uses of a member template whose name resolves to another template at the location of the use.
This option, which defaults to on, enables a warning in the following code:

```
template<typename T> struct set{};
template<typename T> struct trait { typedef const T& type; };
struct Value {
  template<typename T> void set(typename trait<T>::type value) {}
};
void foo() {
  Value v;
  v.set<double>(3.2);
}
```

C++ [basic.lookup.classref] requires this to be an error, but, because it's hard to work around, Clang downgrades it to a warning as an extension.

-Wbind-to-temporary-copy
Warn about an unusable copy constructor when binding a reference to a temporary.

This option enables warnings about binding a reference to a temporary when the temporary doesn't have a usable copy constructor. For example:

```
struct NonCopyable {
  NonCopyable();
private:
  NonCopyable(const NonCopyable&);
};
void foo(const NonCopyable&);
void bar() {
  foo(NonCopyable());  // Disallowed in C++98; allowed in C++11.
}

struct NonCopyable2 {
  NonCopyable2();
  NonCopyable2(NonCopyable2&);
};
void foo(const NonCopyable2&);
void bar() {
  foo(NonCopyable2());  // Disallowed in C++98; allowed in C++11.
}
```

Note that if `NonCopyable2::NonCopyable2()` has a default argument whose instantiation produces a compile error, that error will still be a hard error in C++98 mode even if this warning is turned off.

Options to Control Clang Crash Diagnostics

As unbelievable as it may sound, Clang does crash from time to time. Generally, this only occurs to those living on the bleeding edge. Clang goes to great lengths to assist you in filing a bug report. Specifically, Clang generates preprocessed source file(s) and associated run script(s) upon a crash. These files should be attached to a bug report to ease reproducibility of the failure. Below are the command line options to control the crash diagnostics.

`-fno-crash-diagnostics`
Disable auto-generation of preprocessed source files during a clang crash.

The -fno-crash-diagnostics flag can be helpful for speeding the process of generating a delta reduced test case.

`-fcrash-diagnostics-dir``=<dir>`
Specify where to write the crash diagnostics files; defaults to the usual location for temporary files.

Clang is also capable of generating preprocessed source file(s) and associated run script(s) even without a crash. This is specially useful when trying to generate a reproducer for warnings or errors while using modules.

`-gen-reproducer`
Generates preprocessed source files, a reproducer script and if relevant, a cache containing: built module pcm's and all headers needed to rebuild the same modules.

Options to Emit Optimization Reports

Optimization reports trace, at a high-level, all the major decisions done by compiler transformations. For instance, when the inliner decides to inline function `foo()` into `bar()`, or the loop unroller decides to unroll a loop N times, or the vectorizer decides to vectorize a loop body.

Clang offers a family of flags which the optimizers can use to emit a diagnostic in three cases:

1. When the pass makes a transformation (*-Rpass*).
2. When the pass fails to make a transformation (*-Rpass-missed*).
3. When the pass determines whether or not to make a transformation (*-Rpass-analysis*).

NOTE: Although the discussion below focuses on *-Rpass*, the exact same options apply to *-Rpass-missed* and *-Rpass-analysis*.

Since there are dozens of passes inside the compiler, each of these flags take a regular expression that identifies the name of the pass which should emit the associated diagnostic. For example, to get a report from the inliner, compile the code with:

```
$ clang -O2 -Rpass=inline code.cc -o code
code.cc:4:25: remark: foo inlined into bar [-Rpass=inline]
int bar(int j) { return foo(j, j - 2); }
                        ^
```

Note that remarks from the inliner are identified with *[-Rpass=inline]*. To request a report from every optimization pass, you should use *-Rpass=.** (in fact, you can use any valid POSIX regular expression). However, do not expect a report from every transformation made by the compiler. Optimization remarks do not really make sense outside of the major transformations (e.g., inlining, vectorization, loop optimizations) and not every optimization pass supports this feature.

Note that when using profile-guided optimization information, profile hotness information can be included in the remarks (see -fdiagnostics-show-hotness).

Current limitations

1. Optimization remarks that refer to function names will display the mangled name of the function. Since these remarks are emitted by the back end of the compiler, it does not know anything about the input language, nor its mangling rules.
2. Some source locations are not displayed correctly. The front end has a more detailed source location tracking than the locations included in the debug info (e.g., the front end can locate code inside macro expansions). However, the locations used by *-Rpass* are translated from debug annotations. That translation can be lossy, which results in some remarks having no location information.

Options to Emit Resource Consumption Reports

These are options that report execution time and consumed memory of different compilations steps.

`-fproc-stat-report`=
This option requests driver to print used memory and execution time of each compilation step. The `clang` driver during execution calls different tools, like compiler, assembler, linker etc. With this option the driver reports total execution time, the execution time spent in user mode and peak memory usage of each the called tool. Value of the option specifies where the report is sent to. If it specifies a regular file, the data are saved to this file in CSV format:

```
$ clang -fproc-stat-report=abc foo.c
$ cat abc
clang-11,"/tmp/foo-123456.o",92000,84000,87536
ld,"a.out",900,8000,53568
```

The data on each row represent:

- file name of the tool executable,
- output file name in quotes,
- total execution time in microseconds,
- execution time in user mode in microseconds,
- peak memory usage in Kb.

It is possible to specify this option without any value. In this case statistics are printed on standard output in human readable format:

```
$ clang -fproc-stat-report foo.c
clang-11: output=/tmp/foo-855a8e.o, total=68.000 ms, user=60.000 ms, mem=86920 Kb
ld: output=a.out, total=8.000 ms, user=4.000 ms, mem=52320 Kb
```

The report file specified in the option is locked for write, so this option can be used to collect statistics in parallel builds. The report file is not cleared, new data is appended to it, thus making posible to accumulate build statistics.

You can also use environment variables to control the process statistics reporting. Setting `CC_PRINT_PROC_STAT` to `1` enables the feature, the report goes to stdout in human readable format. Setting `CC_PRINT_PROC_STAT_FILE` to a fully qualified file path makes it report process statistics to the given file in the CSV format. Specifying a relative path will likely lead to multiple files with the same name created in different directories, since the path is relative to a changing working directory.
These environment variables are handy when you need to request the statistics report without changing your build scripts or alter the existing set of compiler options. Note that `-fproc-stat-report` take precedence over `CC_PRINT_PROC_STAT` and `CC_PRINT_PROC_STAT_FILE`.

```
$ export CC_PRINT_PROC_STAT=1
$ export CC_PRINT_PROC_STAT_FILE=~/project-build-proc-stat.csv
$ make
```

Other Options

Clang options that don't fit neatly into other categories.

`-fgnuc-version`=
This flag controls the value of `__GNUC__` and related macros. This flag does not enable or disable any GCC extensions implemented in Clang. Setting the version to zero causes Clang to leave `__GNUC__` and other GNU-namespaced macros, such as `__GXX_WEAK__`, undefined.

`-MV`
When emitting a dependency file, use formatting conventions appropriate for NMake or Jom. Ignored unless another option causes Clang to emit a dependency file.

When Clang emits a dependency file (e.g., you supplied the -M option) most filenames can be written to the file without any special formatting. Different Make tools will treat different sets of characters as "special" and use different conventions for telling the Make tool that the character is actually part of the filename. Normally Clang uses backslash to "escape" a special character, which is the convention used by GNU Make. The -MV option tells Clang to put double-quotes around the entire filename, which is the convention used by NMake and Jom.

Configuration files

Configuration files group command-line options and allow all of them to be specified just by referencing the configuration file. They may be used, for example, to collect options required to tune compilation for particular target, such as -L, -I, -l, –sysroot, codegen options, etc.

The command line option *–config* can be used to specify configuration file in a Clang invocation. For example:

```
clang --config /home/user/cfgs/testing.txt
clang --config debug.cfg
```

If the provided argument contains a directory separator, it is considered as a file path, and options are read from that file. Otherwise the argument is treated as a file name and is searched for sequentially in the directories:

- user directory,
- system directory,
- the directory where Clang executable resides.

Both user and system directories for configuration files are specified during clang build using CMake parameters, CLANG_CONFIG_FILE_USER_DIR and CLANG_CONFIG_FILE_SYSTEM_DIR respectively. The first file found is used. It is an error if the required file cannot be found.

Another way to specify a configuration file is to encode it in executable name. For example, if the Clang executable is named *armv7l-clang* (it may be a symbolic link to *clang*), then Clang will search for file *armv7l.cfg* in the directory where Clang resides.

If a driver mode is specified in invocation, Clang tries to find a file specific for the specified mode. For example, if the executable file is named *x86_64-clang-cl*, Clang first looks for *x86_64-cl.cfg* and if it is not found, looks for *x86_64.cfg*.

If the command line contains options that effectively change target architecture (these are -m32, -EL, and some others) and the configuration file starts with an architecture name, Clang tries to load the configuration file for the effective architecture. For example, invocation:

```
x86_64-clang -m32 abc.c
```

causes Clang search for a file *i368.cfg* first, and if no such file is found, Clang looks for the file *x86_64.cfg*.

The configuration file consists of command-line options specified on one or more lines. Lines composed of whitespace characters only are ignored as well as lines in which the first non-blank character is #. Long options may be split between several lines by a trailing backslash. Here is example of a configuration file:

```
# Several options on line
-c --target=x86_64-unknown-linux-gnu

# Long option split between lines
-I/usr/lib/gcc/x86_64-linux-gnu/5.4.0/../../../../\
include/c++/5.4.0

# other config files may be included
@linux.options
```

Files included by *@file* directives in configuration files are resolved relative to the including file. For example, if a configuration file *~/.llvm/target.cfg* contains the directive *@os/linux.opts*, the file *linux.opts* is searched for in the directory *~/.llvm/os*.

To generate paths relative to the configuration file, the <*CFGDIR*> token may be used. This will expand to the absolute path of the directory containing the configuration file.

In cases where a configuration file is deployed alongside SDK contents, the SDK directory can remain fully portable by using <*CFGDIR*> prefixed paths. In this way, the user may only need to specify a root configuration file with *–config* to establish every aspect of the SDK with the compiler:

```
--target=foo
-isystem <CFGDIR>/include
-L <CFGDIR>/lib
-T <CFGDIR>/ldscripts/link.ld
```

Language and Target-Independent Features

Controlling Errors and Warnings

Clang provides a number of ways to control which code constructs cause it to emit errors and warning messages, and how they are displayed to the console.

Controlling How Clang Displays Diagnostics

When Clang emits a diagnostic, it includes rich information in the output, and gives you fine-grain control over which information is printed. Clang has the ability to print this information, and these are the options that control it:

1. A file/line/column indicator that shows exactly where the diagnostic occurs in your code [-fshow-column, -fshow-source-location].
2. A categorization of the diagnostic as a note, warning, error, or fatal error.
3. A text string that describes what the problem is.
4. An option that indicates how to control the diagnostic (for diagnostics that support it) [-fdiagnostics-show-option].
5. A high-level category for the diagnostic for clients that want to group diagnostics by class (for diagnostics that support it) [-fdiagnostics-show-category].
6. The line of source code that the issue occurs on, along with a caret and ranges that indicate the important locations [-fcaret-diagnostics].
7. “FixIt” information, which is a concise explanation of how to fix the problem (when Clang is certain it knows) [-fdiagnostics-fixit-info].

8. A machine-parsable representation of the ranges involved (off by default) [-fdiagnostics-print-source-range-info].

For more information please see Formatting of Diagnostics.

Diagnostic Mappings

All diagnostics are mapped into one of these 6 classes:

- Ignored
- Note
- Remark
- Warning
- Error
- Fatal

Diagnostic Categories

Though not shown by default, diagnostics may each be associated with a high-level category. This category is intended to make it possible to triage builds that produce a large number of errors or warnings in a grouped way.

Categories are not shown by default, but they can be turned on with the -fdiagnostics-show-category option. When set to “`name`”, the category is printed textually in the diagnostic output. When it is set to “`id`”, a category number is printed. The mapping of category names to category id’s can be obtained by running ‘`clang --print-diagnostic-categories`’.

Controlling Diagnostics via Command Line Flags

TODO: -W flags, -pedantic, etc

Controlling Diagnostics via Pragmas

Clang can also control what diagnostics are enabled through the use of pragmas in the source code. This is useful for turning off specific warnings in a section of source code. Clang supports GCC’s pragma for compatibility with existing source code, as well as several extensions.

The pragma may control any warning that can be used from the command line. Warnings may be set to ignored, warning, error, or fatal. The following example code will tell Clang or GCC to ignore the -Wall warnings:

```
#pragma GCC diagnostic ignored "-Wall"
```

In addition to all of the functionality provided by GCC’s pragma, Clang also allows you to push and pop the current warning state. This is particularly useful when writing a header file that will be compiled by other people, because you don’t know what warning flags they build with.

In the below example **`-Wextra-tokens`** is ignored for only a single line of code, after which the diagnostics return to whatever state had previously existed.

```
#if foo
#endif foo // warning: extra tokens at end of #endif directive

#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wextra-tokens"

#if foo
#endif foo // no warning

#pragma clang diagnostic pop
```

The push and pop pragmas will save and restore the full diagnostic state of the compiler, regardless of how it was set. That means that it is possible to use push and pop around GCC compatible diagnostics and Clang will push and

pop them appropriately, while GCC will ignore the pushes and pops as unknown pragmas. It should be noted that while Clang supports the GCC pragma, Clang and GCC do not support the exact same set of warnings, so even when using GCC compatible #pragmas there is no guarantee that they will have identical behaviour on both compilers.

In addition to controlling warnings and errors generated by the compiler, it is possible to generate custom warning and error messages through the following pragmas:

```
// The following will produce warning messages
#pragma message "some diagnostic message"
#pragma GCC warning "TODO: replace deprecated feature"

// The following will produce an error message
#pragma GCC error "Not supported"
```

These pragmas operate similarly to the `#warning` and `#error` preprocessor directives, except that they may also be embedded into preprocessor macros via the C99 `_Pragma` operator, for example:

```
#define STR(X) #X
#define DEFER(M,...) M(__VA_ARGS__)
#define CUSTOM_ERROR(X) _Pragma(STR(GCC error(X " at line " DEFER(STR,__LINE__))))

CUSTOM_ERROR("Feature not available");
```

Controlling Diagnostics in System Headers

Warnings are suppressed when they occur in system headers. By default, an included file is treated as a system header if it is found in an include path specified by `-isystem`, but this can be overridden in several ways.

The `system_header` pragma can be used to mark the current file as being a system header. No warnings will be produced from the location of the pragma onwards within the same file.

```
#if foo
#endif foo // warning: extra tokens at end of #endif directive

#pragma clang system_header

#if foo
#endif foo // no warning
```

The *–system-header-prefix=* and *–no-system-header-prefix=* command-line arguments can be used to override whether subsets of an include path are treated as system headers. When the name in a `#include` directive is found within a header search path and starts with a system prefix, the header is treated as a system header. The last prefix on the command-line which matches the specified header name takes precedence. For instance:

```
$ clang -Ifoo -isystem bar --system-header-prefix=x/ \
    --no-system-header-prefix=x/y/
```

Here, `#include "x/a.h"` is treated as including a system header, even if the header is found in `foo`, and `#include "x/y/b.h"` is treated as not including a system header, even if the header is found in `bar`.

A `#include` directive which finds a file relative to the current directory is treated as including a system header if the including file is treated as a system header.

Controlling Deprecation Diagnostics in Clang-Provided C Runtime Headers

Clang is responsible for providing some of the C runtime headers that cannot be provided by a platform CRT, such as implementation limits or when compiling in freestanding mode. Define the `_CLANG_DISABLE_CRT_DEPRECATION_WARNINGS` macro prior to including such a C runtime header to disable the deprecation warnings. Note that the C Standard Library headers are allowed to transitively include other standard library headers (see 7.1.2p5), and so the most appropriate use of this macro is to set it within the build system using `-D` or before any include directives in the translation unit.

```
#define _CLANG_DISABLE_CRT_DEPRECATION_WARNINGS
#include <stdint.h>    // Clang CRT deprecation warnings are disabled.
#include <stdatomic.h> // Clang CRT deprecation warnings are disabled.
```

Enabling All Diagnostics

In addition to the traditional `-W` flags, one can enable **all** diagnostics by passing **`-Weverything`**. This works as expected with **`-Werror`**, and also includes the warnings from **`-pedantic`**. Some diagnostics contradict each other, therefore, users of **`-Weverything`** often disable many diagnostics such as *-Wno-c++98-compat* and *-Wno-c++-compat* because they contradict recent C++ standards.

Since **`-Weverything`** enables every diagnostic, we generally don't recommend using it. *-Wall -Wextra* are a better choice for most projects. Using **`-Weverything`** means that updating your compiler is more difficult because you're exposed to experimental diagnostics which might be of lower quality than the default ones. If you do use **`-Weverything`** then we advise that you address all new compiler diagnostics as they get added to Clang, either by fixing everything they find or explicitly disabling that diagnostic with its corresponding *Wno-* option.

Note that when combined with **`-w`** (which disables all warnings), disabling all warnings wins.

Controlling Static Analyzer Diagnostics

While not strictly part of the compiler, the diagnostics from Clang's static analyzer can also be influenced by the user via changes to the source code. See the available annotations and the analyzer's FAQ page for more information.

Precompiled Headers

Precompiled headers are a general approach employed by many compilers to reduce compilation time. The underlying motivation of the approach is that it is common for the same (and often large) header files to be included by multiple source files. Consequently, compile times can often be greatly improved by caching some of the (redundant) work done by a compiler to process headers. Precompiled header files, which represent one of many ways to implement this optimization, are literally files that represent an on-disk cache that contains the vital information necessary to reduce some of the work needed to process a corresponding header file. While details of precompiled headers vary between compilers, precompiled headers have been shown to be highly effective at speeding up program compilation on systems with very large system headers (e.g., macOS).

Generating a PCH File

To generate a PCH file using Clang, one invokes Clang with the *-x <language>-header* option. This mirrors the interface in GCC for generating PCH files:

```
$ gcc -x c-header test.h -o test.h.gch
$ clang -x c-header test.h -o test.h.pch
```

Using a PCH File

A PCH file can then be used as a prefix header when a `-include-pch` option is passed to `clang`:

```
$ clang -include-pch test.h.pch test.c -o test
```

The `clang` driver will check if the PCH file `test.h.pch` is available; if so, the contents of `test.h` (and the files it includes) will be processed from the PCH file. Otherwise, Clang will report an error.

Note

Clang does *not* automatically use PCH files for headers that are directly included within a source file or indirectly via **`-include`**. For example:

```
$ clang -x c-header test.h -o test.h.pch
$ cat test.c
#include "test.h"
$ clang test.c -o test
```

In this example, `clang` will not automatically use the PCH file for `test.h` since `test.h` was included directly in the source file and not specified on the command line using `-include-pch`.

Relocatable PCH Files

It is sometimes necessary to build a precompiled header from headers that are not yet in their final, installed locations. For example, one might build a precompiled header within the build tree that is then meant to be installed alongside the headers. Clang permits the creation of “relocatable” precompiled headers, which are built with a given path (into the build directory) and can later be used from an installed location.

To build a relocatable precompiled header, place your headers into a subdirectory whose structure mimics the installed location. For example, if you want to build a precompiled header for the header `mylib.h` that will be installed into `/usr/include`, create a subdirectory `build/usr/include` and place the header `mylib.h` into that subdirectory. If `mylib.h` depends on other headers, then they can be stored within `build/usr/include` in a way that mimics the installed location.

Building a relocatable precompiled header requires two additional arguments. First, pass the `--relocatable-pch` flag to indicate that the resulting PCH file should be relocatable. Second, pass `-isysroot /path/to/build`, which makes all includes for your library relative to the build directory. For example:

```
# clang -x c-header --relocatable-pch -isysroot /path/to/build /path/to/build/mylib.h mylib.h.pch
```

When loading the relocatable PCH file, the various headers used in the PCH file are found from the system header root. For example, `mylib.h` can be found in `/usr/include/mylib.h`. If the headers are installed in some other system root, the `-isysroot` option can be used provide a different system root from which the headers will be based. For example, `-isysroot /Developer/SDKs/MacOSX10.4u.sdk` will look for `mylib.h` in `/Developer/SDKs/MacOSX10.4u.sdk/usr/include/mylib.h`.

Relocatable precompiled headers are intended to be used in a limited number of cases where the compilation environment is tightly controlled and the precompiled header cannot be generated after headers have been installed.

Controlling Floating Point Behavior

Clang provides a number of ways to control floating point behavior, including with command line options and source pragmas. This section describes the various floating point semantic modes and the corresponding options.

Floating Point Semantic Modes

Mode	Values	
ffp-exception-behavior	{ignore, strict, may_trap}	
fenv_access	{off, on}	(none)
frounding-math	{dynamic, tonearest, downward, upward, towardzero}	
ffp-contract	{on, off, fast, fast-honor-pragmas}	
fdenormal-fp-math	{IEEE, PreserveSign, PositiveZero}	
fdenormal-fp-math-fp32	{IEEE, PreserveSign, PositiveZero}	
fmath-errno	{on, off}	
fhonor-nans	{on, off}	
fhonor-infinities	{on, off}	
fsigned-zeros	{on, off}	
freciprocal-math	{on, off}	

Mode	Values	
allow_approximate_fns	{on, off}	
fassociative-math	{on, off}	

This table describes the option settings that correspond to the three floating point semantic models: precise (the default), strict, and fast.

Floating Point Models

Mode	Precise	Strict	Fast
except_behavior	ignore	strict	ignore
fenv_access	off	on	off
rounding_mode	tonearest	dynamic	tonearest
contract	on	off	fast
denormal_fp_math	IEEE	IEEE	PreserveSign
denormal_fp32_math	IEEE	IEEE	PreserveSign
support_math_errno	on	on	off
no_honor_nans	off	off	on
no_honor_infinities	off	off	on
no_signed_zeros	off	off	on
allow_reciprocal	off	off	on
allow_approximate_fns	off	off	on
allow_reassociation	off	off	on

`-ffast-math`

Enable fast-math mode. This option lets the compiler make aggressive, potentially-lossy assumptions about floating-point math. These include:

- Floating-point math obeys regular algebraic rules for real numbers (e.g. `+` and `*` are associative, `x/y == x * (1/y)`, and `(a + b) * c == a * c + b * c`),
- Operands to floating-point operations are not equal to `NaN` and `Inf`, and
- `+0` and `-0` are interchangeable.

`-ffast-math` also defines the `__FAST_MATH__` preprocessor macro. Some math libraries recognize this macro and change their behavior. With the exception of `-ffp-contract=fast`, using any of the options below to disable any of the individual optimizations in `-ffast-math` will cause `__FAST_MATH__` to no longer be set.

This option implies:

- `-fno-honor-infinities`
- `-fno-honor-nans`
- `-fno-math-errno`
- `-ffinite-math-only`
- `-fassociative-math`
- `-freciprocal-math`
- `-fno-signed-zeros`
- `-fno-trapping-math`

- `-ffp-contract=fast`

`-fdenormal-fp-math`=<value>
Select which denormal numbers the code is permitted to require.
Valid values are:

- `ieee` - IEEE 754 denormal numbers
- `preserve-sign` - the sign of a flushed-to-zero number is preserved in the sign of 0
- `positive-zero` - denormals are flushed to positive zero

Defaults to `ieee`.

-f[no-]strict-float-cast-overflow

When a floating-point value is not representable in a destination integer type, the code has undefined behavior according to the language standard. By default, Clang will not guarantee any particular result in that case. With the 'no-strict' option, Clang will saturate towards the smallest and largest representable integer values instead. NaNs will be converted to zero.

-f[no-]math-errno

Require math functions to indicate errors by setting errno. The default varies by ToolChain. `-fno-math-errno` allows optimizations that might cause standard C math functions to not set `errno`. For example, on some systems, the math function `sqrt` is specified as setting `errno` to `EDOM` when the input is negative. On these systems, the compiler cannot normally optimize a call to `sqrt` to use inline code (e.g. the x86 `sqrtsd` instruction) without additional checking to ensure that `errno` is set appropriately. `-fno-math-errno` permits these transformations.

On some targets, math library functions never set `errno`, and so `-fno-math-errno` is the default. This includes most BSD-derived systems, including Darwin.

-f[no-]trapping-math

Control floating point exception behavior. `-fno-trapping-math` allows optimizations that assume that floating point operations cannot generate traps such as divide-by-zero, overflow and underflow.

- The option `-ftrapping-math` behaves identically to `-ffp-exception-behavior=strict`.
- The option `-fno-trapping-math` behaves identically to `-ffp-exception-behavior=ignore`. This is the default.

`-ffp-contract`=<value>
Specify when the compiler is permitted to form fused floating-point operations, such as fused multiply-add (FMA). Fused operations are permitted to produce more precise results than performing the same operations separately.
The C standard permits intermediate floating-point results within an expression to be computed with more precision than their type would normally allow. This permits operation fusing, and Clang takes advantage of this by default. This behavior can be controlled with the `FP_CONTRACT` and `clang fp contract` pragmas. Please refer to the pragma documentation for a description of how the pragmas interact with this option.
Valid values are:

- `fast` (fuse across statements disregarding pragmas, default for CUDA)
- `on` (fuse in the same statement unless dictated by pragmas, default for languages other than CUDA/HIP)
- `off` (never fuse)
- `fast-honor-pragmas` (fuse across statements unless dictated by pragmas, default for HIP)

-f[no-]honor-infinities

If both `-fno-honor-infinities` and `-fno-honor-nans` are used, has the same effect as specifying `-ffinite-math-only`.

-f[no-]honor-nans

If both `-fno-honor-infinities` and `-fno-honor-nans` are used, has the same effect as specifying `-ffinite-math-only`.

-f[no-]approx-func

Allow certain math function calls (such as `log`, `sqrt`, `pow`, etc) to be replaced with an approximately equivalent set of instructions or alternative math function calls. For example, a `pow(x, 0.25)` may be replaced with `sqrt(sqrt(x))`, despite being an inexact result in cases where `x` is `-0.0` or `-inf`. Defaults to `-fno-approx-func`.

-f[no-]signed-zeros

Allow optimizations that ignore the sign of floating point zeros. Defaults to `-fno-signed-zeros`.

-f[no-]associative-math

Allow floating point operations to be reassociated. Defaults to `-fno-associative-math`.

-f[no-]reciprocal-math

Allow division operations to be transformed into multiplication by a reciprocal. This can be significantly faster than an ordinary division but can also have significantly less precision. Defaults to `-fno-reciprocal-math`.

-f[no-]unsafe-math-optimizations

Allow unsafe floating-point optimizations. Also implies:

- `-fassociative-math`
- `-freciprocal-math`
- `-fno-signed-zeroes`
- `-fno-trapping-math`.

Defaults to `-fno-unsafe-math-optimizations`.

-f[no-]finite-math-only

Allow floating-point optimizations that assume arguments and results are not NaNs or +-Inf. This defines the `__FINITE_MATH_ONLY__` preprocessor macro. Also implies:

- `-fno-honor-infinities`
- `-fno-honor-nans`

Defaults to `-fno-finite-math-only`.

-f[no-]rounding-math

Force floating-point operations to honor the dynamically-set rounding mode by default.

The result of a floating-point operation often cannot be exactly represented in the result type and therefore must be rounded. IEEE 754 describes different rounding modes that control how to perform this rounding, not all of which are supported by all implementations. C provides interfaces (`fesetround` and `fesetenv`) for dynamically controlling the rounding mode, and while it also recommends certain conventions for changing the rounding mode, these conventions are not typically enforced in the ABI. Since the rounding mode changes the numerical result of operations, the compiler must understand something about it in order to optimize floating point operations.

Note that floating-point operations performed as part of constant initialization are formally performed prior to the start of the program and are therefore not subject to the current rounding mode. This includes the initialization of global variables and local `static` variables. Floating-point operations in these contexts will be rounded using `FE_TONEAREST`.

- The option `-fno-rounding-math` allows the compiler to assume that the rounding mode is set to `FE_TONEAREST`. This is the default.
- The option `-frounding-math` forces the compiler to honor the dynamically-set rounding mode. This prevents optimizations which might affect results if the rounding mode changes or is different from the default; for example, it prevents floating-point operations from being reordered across most calls and prevents constant-folding when the result is not exactly representable.

`-ffp-model`=<value>

Specify floating point behavior. `-ffp-model` is an umbrella option that encompasses functionality provided by other, single purpose, floating point options. Valid values are: `precise`, `strict`, and `fast`. Details:

- `precise` Disables optimizations that are not value-safe on floating-point data, although FP contraction (FMA) is enabled (`-ffp-contract=on`). This is the default behavior.

- `strict` Enables `-frounding-math` and `-ffp-exception-behavior=strict`, and disables contractions (FMA). All of the `-ffast-math` enablements are disabled. Enables `STDC FENV_ACCESS`: by default `FENV_ACCESS` is disabled. This option setting behaves as though `#pragma STDC FENV_ACESS ON` appeared at the top of the source file.
- `fast` Behaves identically to specifying both `-ffast-math` and `ffp-contract=fast`

Note: If your command line specifies multiple instances of the `-ffp-model` option, or if your command line option specifies `-ffp-model` and later on the command line selects a floating point option that has the effect of negating part of the `ffp-model` that has been selected, then the compiler will issue a diagnostic warning that the override has occurred.

`-ffp-exception-behavior`=<`value`>
Specify the floating-point exception behavior.
Valid values are: `ignore`, `maytrap`, and `strict`. The default value is `ignore`. Details:

- `ignore` The compiler assumes that the exception status flags will not be read and that floating point exceptions will be masked.
- `maytrap` The compiler avoids transformations that may raise exceptions that would not have been raised by the original code. Constant folding performed by the compiler is exempt from this option.
- `strict` The compiler ensures that all transformations strictly preserve the floating point exception semantics of the original code.

`-ffp-eval-method`=<`value`>
Specify the floating-point evaluation method for intermediate results within a single expression of the code.
Valid values are: `source`, `double`, and `extended`. For 64-bit targets, the default value is `source`. For 32-bit x86 targets however, in the case of NETBSD 6.99.26 and under, the default value is `double`; in the case of NETBSD greater than 6.99.26, with NoSSE, the default value is `extended`, with SSE the default value is `source`. Details:

- `source` The compiler uses the floating-point type declared in the source program as the evaluation method.
- `double` The compiler uses `double` as the floating-point evaluation method for all float expressions of type that is narrower than `double`.
- `extended` The compiler uses `long double` as the floating-point evaluation method for all float expressions of type that is narrower than `long double`.

`-f[no-]protect-parens:`
This option pertains to floating-point types, complex types with floating-point components, and vectors of these types. Some arithmetic expression transformations that are mathematically correct and permissible according to the C and C++ language standards may be incorrect when dealing with floating-point types, such as reassociation and distribution. Further, the optimizer may ignore parentheses when computing arithmetic expressions in circumstances where the parenthesized and unparenthesized expression express the same mathematical value. For example (a+b)+c is the same mathematical value as a+(b+c), but the optimizer is free to evaluate the additions in any order regardless of the parentheses. When enabled, this option forces the optimizer to honor the order of operations with respect to parentheses in all circumstances.
Note that floating-point contraction (option *-ffp-contract=*) is disabled when *-fprotect-parens* is enabled. Also note that in safe floating-point modes, such as *-ffp-model=precise* or *-ffp-model=strict*, this option has no effect because the optimizer is prohibited from making unsafe transformations.

A note about `__FLT_EVAL_METHOD__`

The macro `__FLT_EVAL_METHOD__` will expand to either the value set from the command line option `ffp-eval-method` or to the value from the target info setting. The `__FLT_EVAL_METHOD__` macro cannot expand to the correct evaluation method in the presence of a `#pragma` which alters the evaluation method. An error is issued if `__FLT_EVAL_METHOD__` is expanded inside a scope modified by `#pragma clang fp eval_method`.

A note about Floating Point Constant Evaluation

In C, the only place floating point operations are guaranteed to be evaluated during translation is in the initializers of variables of static storage duration, which are all notionally initialized before the program begins executing (and thus before a non-default floating point environment can be entered). But C++ has many more contexts where floating point constant evaluation occurs. Specifically: for static/thread-local variables, first try evaluating the initializer in a constant context, including in the constant floating point environment (just like in C), and then, if that fails, fall back to emitting runtime code to perform the initialization (which might in general be in a different floating point environment).

Consider this example when compiled with `-frounding-math`

```
constexpr float func_01(float x, float y) {
  return x + y;
}
float V1 = func_01(1.0F, 0x0.000001p0F);
```

The C++ rule is that initializers for static storage duration variables are first evaluated during translation (therefore, in the default rounding mode), and only evaluated at runtime (and therefore in the runtime rounding mode) if the compile-time evaluation fails. This is in line with the C rules; C11 F.8.5 says: *All computation for automatic initialization is done (as if) at execution time; thus, it is affected by any operative modes and raises floating-point exceptions as required by IEC 60559 (provided the state for the FENV_ACCESS pragma is ‘‘on’’). All computation for initialization of objects that have static or thread storage duration is done (as if) at translation time.* C++ generalizes this by adding another phase of initialization (at runtime) if the translation-time initialization fails, but the translation-time evaluation of the initializer of succeeds, it will be treated as a constant initializer.

Controlling Code Generation

Clang provides a number of ways to control code generation. The options are listed below.

-f[no-]sanitize=check1,check2,...

Turn on runtime checks for various forms of undefined or suspicious behavior.

This option controls whether Clang adds runtime checks for various forms of undefined or suspicious behavior, and is disabled by default. If a check fails, a diagnostic message is produced at runtime explaining the problem. The main checks are:

- `-fsanitize=address`: AddressSanitizer, a memory error detector.
- `-fsanitize=thread`: ThreadSanitizer, a data race detector.
- `-fsanitize=memory`: MemorySanitizer, a detector of uninitialized reads. Requires instrumentation of all program code.
- `-fsanitize=undefined`: UndefinedBehaviorSanitizer, a fast and compatible undefined behavior checker.
- `-fsanitize=dataflow`: DataFlowSanitizer, a general data flow analysis.
- `-fsanitize=cfi`: control flow integrity checks. Requires `-flto`.
- `-fsanitize=safe-stack`: safe stack protection against stack-based memory corruption errors.

There are more fine-grained checks available: see the list of specific kinds of undefined behavior that can be detected and the list of control flow integrity schemes.

The `-fsanitize=` argument must also be provided when linking, in order to link to the appropriate runtime library.

It is not possible to combine more than one of the `-fsanitize=address`, `-fsanitize=thread`, and `-fsanitize=memory` checkers in the same program.

-f[no-]sanitize-recover=check1,check2,...

-f[no-]sanitize-recover[=all]

Controls which checks enabled by `-fsanitize=` flag are non-fatal. If the check is fatal, program will halt after the first error of this kind is detected and error report is printed.

By default, non-fatal checks are those enabled by UndefinedBehaviorSanitizer, except for `-fsanitize=return` and `-fsanitize=unreachable`. Some sanitizers may not support recovery (or not support it by default e.g. AddressSanitizer), and always crash the program after the issue is detected.

Note that the `-fsanitize-trap` flag has precedence over this flag. This means that if a check has been configured to trap elsewhere on the command line, or if the check traps by default, this flag will not have any effect unless that sanitizer’s trapping behavior is disabled with `-fno-sanitize-trap`.

For example, if a command line contains the flags `-fsanitize=undefined -fsanitize-trap=undefined`, the flag `-fsanitize-recover=alignment` will have no effect on its own; it will need to be accompanied by `-fno-sanitize-trap=alignment`.

-f[no-]sanitize-trap=check1,check2,...

-f[no-]sanitize-trap[=all]

Controls which checks enabled by the `-fsanitize=` flag trap. This option is intended for use in cases where the sanitizer runtime cannot be used (for instance, when building libc or a kernel module), or where the binary size increase caused by the sanitizer runtime is a concern.

This flag is only compatible with control flow integrity schemes and UndefinedBehaviorSanitizer checks other than `vptr`.

This flag is enabled by default for sanitizers in the `cfi` group.

-fsanitize-ignorelist=/path/to/ignorelist/file
Disable or modify sanitizer checks for objects (source files, functions, variables, types) listed in the file. See Sanitizer special case list for file format description.

-fno-sanitize-ignorelist
Don't use ignorelist file, if it was specified earlier in the command line.

-f[no-]sanitize-coverage=[type,features,...]

Enable simple code coverage in addition to certain sanitizers. See SanitizerCoverage for more details.

-f[no-]sanitize-address-outline-instrumentation

Controls how address sanitizer code is generated. If enabled will always use a function call instead of inlining the code. Turning this option on could reduce the binary size, but might result in a worse run-time performance.

See :doc: *AddressSanitizer* for more details.

-f[no-]sanitize-stats

Enable simple statistics gathering for the enabled sanitizers. See SanitizerStats for more details.

-fsanitize-undefined-trap-on-error
Deprecated alias for `-fsanitize-trap=undefined`.

-fsanitize-cfi-cross-dso
Enable cross-DSO control flow integrity checks. This flag modifies the behavior of sanitizers in the `cfi` group to allow checking of cross-DSO virtual and indirect calls.

-fsanitize-cfi-icall-generalize-pointers
Generalize pointers in return and argument types in function type signatures checked by Control Flow Integrity indirect call checking. See Control Flow Integrity for more details.

-fstrict-vtable-pointers
Enable optimizations based on the strict rules for overwriting polymorphic C++ objects, i.e. the vptr is invariant during an object's lifetime. This enables better devirtualization. Turned off by default, because it is still experimental.

-fwhole-program-vtables
Enable whole-program vtable optimizations, such as single-implementation devirtualization and virtual constant propagation, for classes with hidden LTO visibility. Requires `-flto`.

-fforce-emit-vtables
In order to improve devirtualization, forces emitting of vtables even in modules where it isn't necessary. It causes more inline virtual functions to be emitted.

-fno-assume-sane-operator-new
Don't assume that the C++'s new operator is sane.
This option tells the compiler to do not assume that C++'s global new operator will always return a pointer that does not alias any other pointer when the function returns.

-ftrap-function=[name]
Instruct code generator to emit a function call to the specified function name for `__builtin_trap()`.
LLVM code generator translates `__builtin_trap()` to a trap instruction if it is supported by the target ISA. Otherwise, the builtin is translated into a call to `abort`. If this option is set, then the code generator will always

lower the builtin to a call to the specified function regardless of whether the target ISA has a trap instruction. This option is useful for environments (e.g. deeply embedded) where a trap cannot be properly handled, or when some custom behavior is desired.

-ftls-model=[model]
Select which TLS model to use.
Valid values are: `global-dynamic`, `local-dynamic`, `initial-exec` and `local-exec`. The default value is `global-dynamic`. The compiler may use a different model if the selected model is not supported by the target, or if a more efficient model can be used. The TLS model can be overridden per variable using the `tls_model` attribute.

-femulated-tls
Select emulated TLS model, which overrides all -ftls-model choices.
In emulated TLS mode, all access to TLS variables are converted to calls to __emutls_get_address in the runtime library.

-mhwdiv=[values]
Select the ARM modes (arm or thumb) that support hardware division instructions.
Valid values are: `arm`, `thumb` and `arm,thumb`. This option is used to indicate which mode (arm or thumb) supports hardware division instructions. This only applies to the ARM architecture.

-m[no-]crc
Enable or disable CRC instructions.
This option is used to indicate whether CRC instructions are to be generated. This only applies to the ARM architecture.
CRC instructions are enabled by default on ARMv8.

-mgeneral-regs-only
Generate code which only uses the general purpose registers.
This option restricts the generated code to use general registers only. This only applies to the AArch64 architecture.

-mcompact-branches=[values]
Control the usage of compact branches for MIPSR6.
Valid values are: `never`, `optimal` and `always`. The default value is `optimal` which generates compact branches when a delay slot cannot be filled. `never` disables the usage of compact branches and `always` generates compact branches whenever possible.

-f[no-]max-type-align=[number]

Instruct the code generator to not enforce a higher alignment than the given number (of bytes) when accessing memory via an opaque pointer or reference. This cap is ignored when directly accessing a variable or when the pointee type has an explicit “aligned” attribute.

The value should usually be determined by the properties of the system allocator. Some builtin types, especially vector types, have very high natural alignments; when working with values of those types, Clang usually wants to use instructions that take advantage of that alignment. However, many system allocators do not promise to return memory that is more than 8-byte or 16-byte-aligned. Use this option to limit the alignment that the compiler can assume for an arbitrary pointer, which may point onto the heap.

This option does not affect the ABI alignment of types; the layout of structs and unions and the value returned by the alignof operator remain the same.

This option can be overridden on a case-by-case basis by putting an explicit “aligned” alignment on a struct, union, or typedef. For example:

```
#include <immintrin.h>
// Make an aligned typedef of the AVX-512 16-int vector type.
typedef __v16si __aligned_v16si __attribute__((aligned(64)));

void initialize_vector(__aligned_v16si *v) {
  // The compiler may assume that ‘v’ is 64-byte aligned, regardless of the
  // value of -fmax-type-align.
}
```

-faddrsig, -fno-addrsig
Controls whether Clang emits an address-significance table into the object file. Address-significance tables allow linkers to implement safe ICF without the false positives that can result from other implementation techniques such

as relocation scanning. Address-significance tables are enabled by default on ELF targets when using the integrated assembler. This flag currently only has an effect on ELF targets.

-f[no]-unique-internal-linkage-names

Controls whether Clang emits a unique (best-effort) symbol name for internal linkage symbols. When this option is set, compiler hashes the main source file path from the command line and appends it to all internal symbols. If a program contains multiple objects compiled with the same command-line source file path, the symbols are not guaranteed to be unique. This option is particularly useful in attributing profile information to the correct function when multiple functions with the same private linkage name exist in the binary.

It should be noted that this option cannot guarantee uniqueness and the following is an example where it is not unique when two modules contain symbols with the same private linkage name:

```
$ cd $P/foo && clang -c -funique-internal-linkage-names name_conflict.c
$ cd $P/bar && clang -c -funique-internal-linkage-names name_conflict.c
$ cd $P && clang foo/name_conflict.o && bar/name_conflict.o
```

-fbasic-block-sections=[labels, all, list=<arg>, none]

Controls how Clang emits text sections for basic blocks. With values `all` and `list=<arg>`, each basic block or a subset of basic blocks can be placed in its own unique section. With the “labels” value, normal text sections are emitted, but a `.bb_addr_map` section is emitted which includes address offsets for each basic block in the program, relative to the parent function address.

With the `list=<arg>` option, a file containing the subset of basic blocks that need to placed in unique sections can be specified. The format of the file is as follows. For example, `list=spec.txt` where `spec.txt` is the following:

```
!foo
!!2
!_Z3barv
```

will place the machine basic block with `id 2` in function `foo` in a unique section. It will also place all basic blocks of functions `bar` in unique sections.

Further, section clusters can also be specified using the `list=<arg>` option. For example, `list=spec.txt` where `spec.txt` contains:

```
!foo
!!1 !!3 !!5
!!2 !!4 !!6
```

will create two unique sections for function `foo` with the first containing the odd numbered basic blocks and the second containing the even numbered basic blocks.

Basic block sections allow the linker to reorder basic blocks and enables link-time optimizations like whole program inter-procedural basic block reordering.

Profile Guided Optimization

Profile information enables better optimization. For example, knowing that a branch is taken very frequently helps the compiler make better decisions when ordering basic blocks. Knowing that a function `foo` is called more frequently than another function `bar` helps the inliner. Optimization levels `-O2` and above are recommended for use of profile guided optimization.

Clang supports profile guided optimization with two different kinds of profiling. A sampling profiler can generate a profile with very low runtime overhead, or you can build an instrumented version of the code that collects more detailed profile information. Both kinds of profiles can provide execution counts for instructions in the code and information on branches taken and function invocation.

Regardless of which kind of profiling you use, be careful to collect profiles by running your code with inputs that are representative of the typical behavior. Code that is not exercised in the profile will be optimized as if it is unimportant, and the compiler may make poor optimization choices for code that is disproportionately used while profiling.

Differences Between Sampling and Instrumentation

Although both techniques are used for similar purposes, there are important differences between the two:

1. Profile data generated with one cannot be used by the other, and there is no conversion tool that can convert one to the other. So, a profile generated via `-fprofile-instr-generate` must be used with `-fprofile-instr-use`. Similarly, sampling profiles generated by external profilers must be converted and used with `-fprofile-sample-use`.
2. Instrumentation profile data can be used for code coverage analysis and optimization.
3. Sampling profiles can only be used for optimization. They cannot be used for code coverage analysis. Although it would be technically possible to use sampling profiles for code coverage, sample-based profiles are too coarse-grained for code coverage purposes; it would yield poor results.
4. Sampling profiles must be generated by an external tool. The profile generated by that tool must then be converted into a format that can be read by LLVM. The section on sampling profilers describes one of the supported sampling profile formats.

Using Sampling Profilers

Sampling profilers are used to collect runtime information, such as hardware counters, while your application executes. They are typically very efficient and do not incur a large runtime overhead. The sample data collected by the profiler can be used during compilation to determine what the most executed areas of the code are.

Using the data from a sample profiler requires some changes in the way a program is built. Before the compiler can use profiling information, the code needs to execute under the profiler. The following is the usual build cycle when using sample profilers for optimization:

1. Build the code with source line table information. You can use all the usual build flags that you always build your application with. The only requirement is that you add `-gline-tables-only` or `-g` to the command line. This is important for the profiler to be able to map instructions back to source line locations.

   ```
   $ clang++ -O2 -gline-tables-only code.cc -o code
   ```

2. Run the executable under a sampling profiler. The specific profiler you use does not really matter, as long as its output can be converted into the format that the LLVM optimizer understands. Currently, there exists a conversion tool for the Linux Perf profiler (https://perf.wiki.kernel.org/), so these examples assume that you are using Linux Perf to profile your code.

   ```
   $ perf record -b ./code
   ```

   Note the use of the `-b` flag. This tells Perf to use the Last Branch Record (LBR) to record call chains. While this is not strictly required, it provides better call information, which improves the accuracy of the profile data.

3. Convert the collected profile data to LLVM's sample profile format. This is currently supported via the AutoFDO converter `create_llvm_prof`. It is available at https://github.com/google/autofdo. Once built and installed, you can convert the `perf.data` file to LLVM using the command:

   ```
   $ create_llvm_prof --binary=./code --out=code.prof
   ```

   This will read `perf.data` and the binary file `./code` and emit the profile data in `code.prof`. Note that if you ran `perf` without the `-b` flag, you need to use `--use_lbr=false` when calling `create_llvm_prof`.

4. Build the code again using the collected profile. This step feeds the profile back to the optimizers. This should result in a binary that executes faster than the original one. Note that you are not required to build the code with the exact same arguments that you used in the first step. The only requirement is that you build the code with `-gline-tables-only` and `-fprofile-sample-use`.

   ```
   $ clang++ -O2 -gline-tables-only -fprofile-sample-use=code.prof code.cc -o code
   ```

Sample Profile Formats

Since external profilers generate profile data in a variety of custom formats, the data generated by the profiler must be converted into a format that can be read by the backend. LLVM supports three different sample profile formats:

1. ASCII text. This is the easiest one to generate. The file is divided into sections, which correspond to each of the functions with profile information. The format is described below. It can also be generated from the binary or gcov formats using the `llvm-profdata` tool.
2. Binary encoding. This uses a more efficient encoding that yields smaller profile files. This is the format generated by the `create_llvm_prof` tool in https://github.com/google/autofdo.
3. GCC encoding. This is based on the gcov format, which is accepted by GCC. It is only interesting in environments where GCC and Clang co-exist. This encoding is only generated by the `create_gcov` tool in https://github.com/google/autofdo. It can be read by LLVM and `llvm-profdata`, but it cannot be generated by either.

If you are using Linux Perf to generate sampling profiles, you can use the conversion tool `create_llvm_prof` described in the previous section. Otherwise, you will need to write a conversion tool that converts your profiler's native format into one of these three.

Sample Profile Text Format

This section describes the ASCII text format for sampling profiles. It is, arguably, the easiest one to generate. If you are interested in generating any of the other two, consult the `ProfileData` library in LLVM's source tree (specifically, `include/llvm/ProfileData/SampleProfReader.h`).

```
function1:total_samples:total_head_samples
 offset1[.discriminator]: number_of_samples [fn1:num fn2:num ... ]
 offset2[.discriminator]: number_of_samples [fn3:num fn4:num ... ]
 ...
 offsetN[.discriminator]: number_of_samples [fn5:num fn6:num ... ]
 offsetA[.discriminator]: fnA:num_of_total_samples
  offsetA1[.discriminator]: number_of_samples [fn7:num fn8:num ... ]
  offsetA1[.discriminator]: number_of_samples [fn9:num fn10:num ... ]
  offsetB[.discriminator]: fnB:num_of_total_samples
   offsetB1[.discriminator]: number_of_samples [fn11:num fn12:num ... ]
```

This is a nested tree in which the indentation represents the nesting level of the inline stack. There are no blank lines in the file. And the spacing within a single line is fixed. Additional spaces will result in an error while reading the file.

Any line starting with the '#' character is completely ignored.

Inlined calls are represented with indentation. The Inline stack is a stack of source locations in which the top of the stack represents the leaf function, and the bottom of the stack represents the actual symbol to which the instruction belongs.

Function names must be mangled in order for the profile loader to match them in the current translation unit. The two numbers in the function header specify how many total samples were accumulated in the function (first number), and the total number of samples accumulated in the prologue of the function (second number). This head sample count provides an indicator of how frequently the function is invoked.

There are two types of lines in the function body.

- Sampled line represents the profile information of a source location. `offsetN[.discriminator]: number_of_samples [fn5:num fn6:num ... ]`
- Callsite line represents the profile information of an inlined callsite. `offsetA[.discriminator]: fnA:num_of_total_samples`

Each sampled line may contain several items. Some are optional (marked below):

a. Source line offset. This number represents the line number in the function where the sample was collected. The line number is always relative to the line where symbol of the function is defined. So, if the function has its header at line 280, the offset 13 is at line 293 in the file.

   Note that this offset should never be a negative number. This could happen in cases like macros. The debug machinery will register the line number at the point of macro expansion. So, if the macro was expanded in a line before the start of the function, the profile converter should emit a 0 as the offset (this means that the optimizers will not be able to associate a meaningful weight to the instructions in the macro).

b. [OPTIONAL] Discriminator. This is used if the sampled program was compiled with DWARF discriminator support (http://wiki.dwarfstd.org/index.php?title=Path_Discriminators). DWARF discriminators are unsigned integer values that allow the compiler to distinguish between multiple execution paths on the same source line location.

   For example, consider the line of code `if (cond) foo(); else bar();`. If the predicate `cond` is true 80% of the time, then the edge into function `foo` should be considered to be taken most of the time. But both calls to `foo` and `bar` are at the same source line, so a sample count at that line is not sufficient. The compiler needs to know which part of that line is taken more frequently.

   This is what discriminators provide. In this case, the calls to `foo` and `bar` will be at the same line, but will have different discriminator values. This allows the compiler to correctly set edge weights into `foo` and `bar`.

c. Number of samples. This is an integer quantity representing the number of samples collected by the profiler at this source location.

d. [OPTIONAL] Potential call targets and samples. If present, this line contains a call instruction. This models both direct and number of samples. For example,

```
130: 7  foo:3  bar:2  baz:7
```

   The above means that at relative line offset 130 there is a call instruction that calls one of `foo()`, `bar()` and `baz()`, with `baz()` being the relatively more frequently called target.

As an example, consider a program with the call chain `main -> foo -> bar`. When built with optimizations enabled, the compiler may inline the calls to `bar` and `foo` inside `main`. The generated profile could then be something like this:

```
main:35504:0
1: _Z3foov:35504
  2: _Z32bari:31977
  1.1: 31977
2: 0
```

This profile indicates that there were a total of 35,504 samples collected in main. All of those were at line 1 (the call to `foo`). Of those, 31,977 were spent inside the body of `bar`. The last line of the profile (`2: 0`) corresponds to line 2 inside `main`. No samples were collected there.

Profiling with Instrumentation

Clang also supports profiling via instrumentation. This requires building a special instrumented version of the code and has some runtime overhead during the profiling, but it provides more detailed results than a sampling profiler. It also provides reproducible results, at least to the extent that the code behaves consistently across runs.

Here are the steps for using profile guided optimization with instrumentation:

1. Build an instrumented version of the code by compiling and linking with the `-fprofile-instr-generate` option.

```
$ clang++ -O2 -fprofile-instr-generate code.cc -o code
```

2. Run the instrumented executable with inputs that reflect the typical usage. By default, the profile data will be written to a `default.profraw` file in the current directory. You can override that default by using option `-fprofile-instr-generate=` or by setting the `LLVM_PROFILE_FILE` environment variable to specify an alternate file. If non-default file name is specified by both the environment variable and the command line option, the environment variable takes precedence. The file name pattern specified can include different modifiers: `%p`, `%h`, and `%m`.

   Any instance of `%p` in that file name will be replaced by the process ID, so that you can easily distinguish the profile output from multiple runs.

```
$ LLVM_PROFILE_FILE="code-%p.profraw" ./code
```

The modifier %h can be used in scenarios where the same instrumented binary is run in multiple different host machines dumping profile data to a shared network based storage. The %h specifier will be substituted with the hostname so that profiles collected from different hosts do not clobber each other.

While the use of %p specifier can reduce the likelihood for the profiles dumped from different processes to clobber each other, such clobbering can still happen because of the pid re-use by the OS. Another side-effect of using %p is that the storage requirement for raw profile data files is greatly increased. To avoid issues like this, the %m specifier can used in the profile name. When this specifier is used, the profiler runtime will substitute %m with a unique integer identifier associated with the instrumented binary. Additionally, multiple raw profiles dumped from different processes that share a file system (can be on different hosts) will be automatically merged by the profiler runtime during the dumping. If the program links in multiple instrumented shared libraries, each library will dump the profile data into its own profile data file (with its unique integer id embedded in the profile name). Note that the merging enabled by %m is for raw profile data generated by profiler runtime. The resulting merged “raw” profile data file still needs to be converted to a different format expected by the compiler (see step 3 below).

```
$ LLVM_PROFILE_FILE="code-%m.profraw" ./code
```

3. Combine profiles from multiple runs and convert the “raw” profile format to the input expected by clang. Use the merge command of the llvm-profdata tool to do this.

```
$ llvm-profdata merge -output=code.profdata code-*.profraw
```

Note that this step is necessary even when there is only one “raw” profile, since the merge operation also changes the file format.

4. Build the code again using the -fprofile-instr-use option to specify the collected profile data.

```
$ clang++ -O2 -fprofile-instr-use=code.profdata code.cc -o code
```

You can repeat step 4 as often as you like without regenerating the profile. As you make changes to your code, clang may no longer be able to use the profile data. It will warn you when this happens.

Profile generation using an alternative instrumentation method can be controlled by the GCC-compatible flags -fprofile-generate and -fprofile-use. Although these flags are semantically equivalent to their GCC counterparts, they *do not* handle GCC-compatible profiles. They are only meant to implement GCC’s semantics with respect to profile creation and use. Flag -fcs-profile-generate also instruments programs using the same instrumentation method as -fprofile-generate.

-fprofile-generate[=<dirname>]

The -fprofile-generate and -fprofile-generate= flags will use an alternative instrumentation method for profile generation. When given a directory name, it generates the profile file default_%m.profraw in the directory named dirname if specified. If dirname does not exist, it will be created at runtime. %m specifier will be substituted with a unique id documented in step 2 above. In other words, with -fprofile-generate[=<dirname>] option, the “raw” profile data automatic merging is turned on by default, so there will no longer any risk of profile clobbering from different running processes. For example,

```
$ clang++ -O2 -fprofile-generate=yyy/zzz code.cc -o code
```

When code is executed, the profile will be written to the file yyy/zzz/default_xxxx.profraw.

To generate the profile data file with the compiler readable format, the llvm-profdata tool can be used with the profile directory as the input:

```
$ llvm-profdata merge -output=code.profdata yyy/zzz/
```

If the user wants to turn off the auto-merging feature, or simply override the the profile dumping path specified at command line, the environment variable LLVM_PROFILE_FILE can still be used to override the directory and filename for the profile file at runtime.

-fcs-profile-generate[=<dirname>]

The -fcs-profile-generate and -fcs-profile-generate= flags will use the same instrumentation method, and generate the same profile as in the -fprofile-generate and -fprofile-generate= flags. The difference is that the instrumentation is performed after inlining so that the resulted profile has a better context sensitive information. They cannot be used together with -fprofile-generate and -fprofile-generate=

flags. They are typically used in conjunction with `-fprofile-use` flag. The profile generated by `-fcs-profile-generate` and `-fprofile-generate` can be merged by llvm-profdata. A use example:

```
$ clang++ -O2 -fprofile-generate=yyy/zzz code.cc -o code
$ ./code
$ llvm-profdata merge -output=code.profdata yyy/zzz/
```

The first few steps are the same as that in `-fprofile-generate` compilation. Then perform a second round of instrumentation.

```
$ clang++ -O2 -fprofile-use=code.profdata -fcs-profile-generate=sss/ttt \
  -o cs_code
$ ./cs_code
$ llvm-profdata merge -output=cs_code.profdata sss/ttt code.profdata
```

The resulted `cs_code.prodata` combines `code.profdata` and the profile generated from binary `cs_code`. Profile `cs_code.profata` can be used by `-fprofile-use` compilation.

```
$ clang++ -O2 -fprofile-use=cs_code.profdata
```

The above command will read both profiles to the compiler at the identical point of instrumentations.

`-fprofile-use``[=<pathname>]`
Without any other arguments, `-fprofile-use` behaves identically to `-fprofile-instr-use`. Otherwise, if `pathname` is the full path to a profile file, it reads from that file. If `pathname` is a directory name, it reads from `pathname/default.profdata`.

`-fprofile-update``[=<method>]`
Unless `-fsanitize=thread` is specified, the default is `single`, which uses non-atomic increments. The counters can be inaccurate under thread contention. `atomic` uses atomic increments which is accurate but has overhead. `prefer-atomic` will be transformed to `atomic` when supported by the target, or `single` otherwise. This option currently works with `-fprofile-arcs` and `-fprofile-instr-generate`, but not with `-fprofile-generate`.

Disabling Instrumentation

In certain situations, it may be useful to disable profile generation or use for specific files in a build, without affecting the main compilation flags used for the other files in the project.

In these cases, you can use the flag `-fno-profile-instr-generate` (or `-fno-profile-generate`) to disable profile generation, and `-fno-profile-instr-use` (or `-fno-profile-use`) to disable profile use.

Note that these flags should appear after the corresponding profile flags to have an effect.

Note

When none of the translation units inside a binary is instrumented, in the case of Fuchsia the profile runtime will not be linked into the binary and no profile will be produced, while on other platforms the profile runtime will be linked and profile will be produced but there will not be any counters.

Instrumenting only selected files or functions

Sometimes it's useful to only instrument certain files or functions. For example in automated testing infrastructure, it may be desirable to only instrument files or functions that were modified by a patch to reduce the overhead of instrumenting a full system.

This can be done using the `-fprofile-list` option.

`-fprofile-list``=<pathname>`
This option can be used to apply profile instrumentation only to selected files or functions. `pathname` should point to a file in the Sanitizer special case list format which selects which files and functions to instrument.

```
$ echo "fun:test" > fun.list
$ clang++ -O2 -fprofile-instr-generate -fprofile-list=fun.list code.cc -o code
```

The option can be specified multiple times to pass multiple files.

```
$ echo "!fun:*test*" > fun.list
$ echo "src:code.cc" > src.list
% clang++ -O2 -fprofile-instr-generate -fcoverage-mapping -fprofile-list=fun.list -fprofile-list=code.list code.cc -o code
```

To filter individual functions or entire source files using `fun:<name>` or `src:<file>` respectively. To exclude a function or a source file, use `!fun:<name>` or `!src:<file>` respectively. The format also supports wildcard expansion. The compiler generated functions are assumed to be located in the main source file. It is also possible to restrict the filter to a particular instrumentation type by using a named section.

```
# all functions whose name starts with foo will be instrumented.
fun:foo*

# except for foo1 which will be excluded from instrumentation.
!fun:foo1

# every function in path/to/foo.cc will be instrumented.
src:path/to/foo.cc

# bar will be instrumented only when using backend instrumentation.
# Recognized section names are clang, llvm and csllvm.
[llvm]
fun:bar
```

When the file contains only excludes, all files and functions except for the excluded ones will be instrumented. Otherwise, only the files and functions specified will be instrumented.

Profile remapping

When the program is compiled after a change that affects many symbol names, pre-existing profile data may no longer match the program. For example:

- switching from libstdc++ to libc++ will result in the mangled names of all functions taking standard library types to change
- renaming a widely-used type in C++ will result in the mangled names of all functions that have parameters involving that type to change
- moving from a 32-bit compilation to a 64-bit compilation may change the underlying type of `size_t` and similar types, resulting in changes to manglings

Clang allows use of a profile remapping file to specify that such differences in mangled names should be ignored when matching the profile data against the program.

`-fprofile-remapping-file`=<file>
Specifies a file containing profile remapping information, that will be used to match mangled names in the profile data to mangled names in the program.

The profile remapping file is a text file containing lines of the form

```
fragmentkind fragment1 fragment2
```

where `fragmentkind` is one of `name`, `type`, or `encoding`, indicating whether the following mangled name fragments are <name>s, <type>s, or <encoding>s, respectively. Blank lines and lines starting with # are ignored.

For convenience, built-in <substitution>s such as `St` and `Ss` are accepted as <name>s (even though they technically are not <name>s).

For example, to specify that `absl::string_view` and `std::string_view` should be treated as equivalent when matching profile data, the following remapping file could be used:

```
# absl::string_view is considered equivalent to std::string_view
type N4absl11string_viewE St17basic_string_viewIcSt11char_traitsIcEE

# std:: might be std::__1:: in libc++ or std::__cxx11:: in libstdc++
name 3std St3__1
name 3std St7__cxx11
```

Matching profile data using a profile remapping file is supported on a best-effort basis. For example, information regarding indirect call targets is currently not remapped. For best results, you are encouraged to generate new profile data matching the updated program, or to remap the profile data using the `llvm-cxxmap` and `llvm-profdata merge` tools.

> **Note**
>
> Profile data remapping is currently only supported for C++ mangled names following the Itanium C++ ABI mangling scheme. This covers all C++ targets supported by Clang other than Windows.

GCOV-based Profiling

GCOV is a test coverage program, it helps to know how often a line of code is executed. When instrumenting the code with `--coverage` option, some counters are added for each edge linking basic blocks.

At compile time, gcno files are generated containing information about blocks and edges between them. At runtime the counters are incremented and at exit the counters are dumped in gcda files.

The tool `llvm-cov gcov` will parse gcno, gcda and source files to generate a report `.c.gcov`.

`-fprofile-filter-files``=[regexes]`
Define a list of regexes separated by a semi-colon. If a file name matches any of the regexes then the file is instrumented.

```
$ clang --coverage -fprofile-filter-files=".*\.c$" foo.c
```

For example, this will only instrument files finishing with `.c`, skipping `.h` files.

`-fprofile-exclude-files``=[regexes]`
Define a list of regexes separated by a semi-colon. If a file name doesn't match all the regexes then the file is instrumented.

```
$ clang --coverage -fprofile-exclude-files="^/usr/include/.*$" foo.c
```

For example, this will instrument all the files except the ones in `/usr/include`.

If both options are used then a file is instrumented if its name matches any of the regexes from `-fprofile-filter-list` and doesn't match all the regexes from `-fprofile-exclude-list`.

```
$ clang --coverage -fprofile-exclude-files="^/usr/include/.*$" \
        -fprofile-filter-files="^/usr/.*$"
```

In that case `/usr/foo/oof.h` is instrumented since it matches the filter regex and doesn't match the exclude regex, but `/usr/include/foo.h` doesn't since it matches the exclude regex.

Controlling Debug Information

Controlling Size of Debug Information

Debug info kind generated by Clang can be set by one of the flags listed below. If multiple flags are present, the last one is used.

`-g0`
Don't generate any debug info (default).

`-gline-tables-only`
Generate line number tables only.
This kind of debug info allows to obtain stack traces with function names, file names and line numbers (by such tools as `gdb` or `addr2line`). It doesn't contain any other data (e.g. description of local variables or function parameters).

`-fstandalone-debug`
Clang supports a number of optimizations to reduce the size of debug information in the binary. They work based on the assumption that the debug type information can be spread out over multiple compilation units. For instance,

Clang will not emit type definitions for types that are not needed by a module and could be replaced with a forward declaration. Further, Clang will only emit type info for a dynamic C++ class in the module that contains the vtable for the class.
The **-fstandalone-debug** option turns off these optimizations. This is useful when working with 3rd-party libraries that don't come with debug information. Note that Clang will never emit type information for types that are not referenced at all by the program.

`-fno-standalone-debug`
On Darwin **-fstandalone-debug** is enabled by default. The **-fno-standalone-debug** option can be used to get to turn on the vtable-based optimization described above.

`-fuse-ctor-homing`
This optimization is similar to the optimizations that are enabled as part of -fno-standalone-debug. Here, Clang only emits type info for a non-trivial, non-aggregate C++ class in the modules that contain a definition of one of its constructors. This relies on the additional assumption that all classes that are not trivially constructible have a non-trivial constructor that is used somewhere. The negation, -fno-use-ctor-homing, ensures that constructor homing is not used.
This flag is not enabled by default, and needs to be used with -cc1 or -Xclang.

`-g`
Generate complete debug info.

`-feliminate-unused-debug-types`
By default, Clang does not emit type information for types that are defined but not used in a program. To retain the debug info for these unused types, the negation **-fno-eliminate-unused-debug-types** can be used.

Controlling Macro Debug Info Generation

Debug info for C preprocessor macros increases the size of debug information in the binary. Macro debug info generated by Clang can be controlled by the flags listed below.

`-fdebug-macro`
Generate debug info for preprocessor macros. This flag is discarded when **-g0** is enabled.

`-fno-debug-macro`
Do not generate debug info for preprocessor macros (default).

Controlling Debugger "Tuning"

While Clang generally emits standard DWARF debug info (http://dwarfstd.org), different debuggers may know how to take advantage of different specific DWARF features. You can "tune" the debug info for one of several different debuggers.

`-ggdb, -glldb, -gsce, -gdbx`
Tune the debug info for the `gdb`, `lldb`, Sony PlayStation® debugger, or `dbx`, respectively. Each of these options implies **-g**. (Therefore, if you want both **-gline-tables-only** and debugger tuning, the tuning option must come first.)

Controlling LLVM IR Output

Controlling Value Names in LLVM IR

Emitting value names in LLVM IR increases the size and verbosity of the IR. By default, value names are only emitted in assertion-enabled builds of Clang. However, when reading IR it can be useful to re-enable the emission of value names to improve readability.

`-fdiscard-value-names`
Discard value names when generating LLVM IR.

`-fno-discard-value-names`
Do not discard value names when generating LLVM IR. This option can be used to re-enable names for release builds of Clang.

Comment Parsing Options

Clang parses Doxygen and non-Doxygen style documentation comments and attaches them to the appropriate declaration nodes. By default, it only parses Doxygen-style comments and ignores ordinary comments starting with `//` and `/*`.

`-Wdocumentation`
Emit warnings about use of documentation comments. This warning group is off by default.
This includes checking that `\param` commands name parameters that actually present in the function signature, checking that `\returns` is used only on functions that actually return a value etc.

`-Wno-documentation-unknown-command`
Don't warn when encountering an unknown Doxygen command.

`-fparse-all-comments`
Parse all comments as documentation comments (including ordinary comments starting with `//` and `/*`).

`-fcomment-block-commands``=[commands]`
Define custom documentation commands as block commands. This allows Clang to construct the correct AST for these custom commands, and silences warnings about unknown commands. Several commands must be separated by a comma *without trailing space*; e.g. `-fcomment-block-commands=foo,bar` defines custom commands `\foo` and `\bar`.
It is also possible to use `-fcomment-block-commands` several times; e.g. `-fcomment-block-commands=foo -fcomment-block-commands=bar` does the same as above.

C Language Features

The support for standard C in clang is feature-complete except for the C99 floating-point pragmas.

Extensions supported by clang

See Clang Language Extensions.

Differences between various standard modes

clang supports the -std option, which changes what language mode clang uses. The supported modes for C are c89, gnu89, c94, c99, gnu99, c11, gnu11, c17, gnu17, c2x, gnu2x, and various aliases for those modes. If no -std option is specified, clang defaults to gnu17 mode. Many C99 and C11 features are supported in earlier modes as a conforming extension, with a warning. Use `-pedantic-errors` to request an error if a feature from a later standard revision is used in an earlier mode.

Differences between all `c*` and `gnu*` modes:

- `c*` modes define "`__STRICT_ANSI__`".
- Target-specific defines not prefixed by underscores, like `linux`, are defined in `gnu*` modes.
- Trigraphs default to being off in `gnu*` modes; they can be enabled by the `-trigraphs` option.
- The parser recognizes `asm` and `typeof` as keywords in `gnu*` modes; the variants `__asm__` and `__typeof__` are recognized in all modes.
- The parser recognizes `inline` as a keyword in `gnu*` mode, in addition to recognizing it in the `*99` and later modes for which it is part of the ISO C standard. The variant `__inline__` is recognized in all modes.
- The Apple "blocks" extension is recognized by default in `gnu*` modes on some platforms; it can be enabled in any mode with the `-fblocks` option.

Differences between `*89` and `*94` modes:

- Digraphs are not recognized in c89 mode.

Differences between `*94` and `*99` modes:

- The `*99` modes default to implementing `inline` / `__inline__` as specified in C99, while the `*89` modes implement the GNU version. This can be overridden for individual functions with the `__gnu_inline__` attribute.

- The scope of names defined inside a `for`, `if`, `switch`, `while`, or `do` statement is different. (example: `if ((struct x {int x;}*)0) {}`.)
- `__STDC_VERSION__` is not defined in `*89` modes.
- `inline` is not recognized as a keyword in `c89` mode.
- `restrict` is not recognized as a keyword in `*89` modes.
- Commas are allowed in integer constant expressions in `*99` modes.
- Arrays which are not lvalues are not implicitly promoted to pointers in `*89` modes.
- Some warnings are different.

Differences between `*99` and `*11` modes:

- Warnings for use of C11 features are disabled.
- `__STDC_VERSION__` is defined to `201112L` rather than `199901L`.

Differences between `*11` and `*17` modes:

- `__STDC_VERSION__` is defined to `201710L` rather than `201112L`.

GCC extensions not implemented yet

clang tries to be compatible with gcc as much as possible, but some gcc extensions are not implemented yet:

- clang does not support decimal floating point types (`_Decimal32` and friends) yet.
- clang does not support nested functions; this is a complex feature which is infrequently used, so it is unlikely to be implemented anytime soon. In C++11 it can be emulated by assigning lambda functions to local variables, e.g:

```
auto const local_function = [&](int parameter) {
  // Do something
};
...
local_function(1);
```

- clang only supports global register variables when the register specified is non-allocatable (e.g. the stack pointer). Support for general global register variables is unlikely to be implemented soon because it requires additional LLVM backend support.
- clang does not support static initialization of flexible array members. This appears to be a rarely used extension, but could be implemented pending user demand.
- clang does not support `__builtin_va_arg_pack`/`__builtin_va_arg_pack_len`. This is used rarely, but in some potentially interesting places, like the glibc headers, so it may be implemented pending user demand. Note that because clang pretends to be like GCC 4.2, and this extension was introduced in 4.3, the glibc headers will not try to use this extension with clang at the moment.
- clang does not support the gcc extension for forward-declaring function parameters; this has not shown up in any real-world code yet, though, so it might never be implemented.

This is not a complete list; if you find an unsupported extension missing from this list, please send an e-mail to cfe-dev. This list currently excludes C++; see C++ Language Features. Also, this list does not include bugs in mostly-implemented features; please see the bug tracker for known existing bugs (FIXME: Is there a section for bug-reporting guidelines somewhere?).

Intentionally unsupported GCC extensions

- clang does not support the gcc extension that allows variable-length arrays in structures. This is for a few reasons: one, it is tricky to implement, two, the extension is completely undocumented, and three, the extension appears to be rarely used. Note that clang *does* support flexible array members (arrays with a zero or unspecified size at the end of a structure).

- GCC accepts many expression forms that are not valid integer constant expressions in bit-field widths, enumerator constants, case labels, and in array bounds at global scope. Clang also accepts additional expression forms in these contexts, but constructs that GCC accepts due to simplifications GCC performs while parsing, such as `x - x` (where `x` is a variable) will likely never be accepted by Clang.
- clang does not support `__builtin_apply` and friends; this extension is extremely obscure and difficult to implement reliably.

Microsoft extensions

clang has support for many extensions from Microsoft Visual C++. To enable these extensions, use the `-fms-extensions` command-line option. This is the default for Windows targets. Clang does not implement every pragma or declspec provided by MSVC, but the popular ones, such as `__declspec(dllexport)` and `#pragma comment(lib)` are well supported.

clang has a `-fms-compatibility` flag that makes clang accept enough invalid C++ to be able to parse most Microsoft headers. For example, it allows unqualified lookup of dependent base class members, which is a common compatibility issue with clang. This flag is enabled by default for Windows targets.

`-fdelayed-template-parsing` lets clang delay parsing of function template definitions until the end of a translation unit. This flag is enabled by default for Windows targets.

For compatibility with existing code that compiles with MSVC, clang defines the `_MSC_VER` and `_MSC_FULL_VER` macros. These default to the values of 1800 and 180000000 respectively, making clang look like an early release of Visual C++ 2013. The `-fms-compatibility-version=` flag overrides these values. It accepts a dotted version tuple, such as 19.00.23506. Changing the MSVC compatibility version makes clang behave more like that version of MSVC. For example, `-fms-compatibility-version=19` will enable C++14 features and define `char16_t` and `char32_t` as builtin types.

C++ Language Features

clang fully implements all of standard C++98 except for exported templates (which were removed in C++11), all of standard C++11, C++14, and C++17, and most of C++20.

See the C++ support in Clang page for detailed information on C++ feature support across Clang versions.

Controlling implementation limits

`-fbracket-depth``=N`
Sets the limit for nested parentheses, brackets, and braces to N. The default is 256.

`-fconstexpr-depth``=N`
Sets the limit for recursive constexpr function invocations to N. The default is 512.

`-fconstexpr-steps``=N`
Sets the limit for the number of full-expressions evaluated in a single constant expression evaluation. The default is 1048576.

`-ftemplate-depth``=N`
Sets the limit for recursively nested template instantiations to N. The default is 1024.

`-foperator-arrow-depth``=N`
Sets the limit for iterative calls to 'operator->' functions to N. The default is 256.

Objective-C Language Features

Objective-C++ Language Features

OpenMP Features

Clang supports all OpenMP 4.5 directives and clauses. See OpenMP Support for additional details.

Use *-fopenmp* to enable OpenMP. Support for OpenMP can be disabled with *-fno-openmp*.

Use *-fopenmp-simd* to enable OpenMP simd features only, without linking the runtime library; for combined constructs (e.g. `#pragma omp parallel for simd`) the non-simd directives and clauses will be ignored. This can be disabled with *-fno-openmp-simd*.

Controlling implementation limits

`-fopenmp-use-tls`
Controls code generation for OpenMP threadprivate variables. In presence of this option all threadprivate variables are generated the same way as thread local variables, using TLS support. If *-fno-openmp-use-tls* is provided or target does not support TLS, code generation for threadprivate variables relies on OpenMP runtime library.

OpenCL Features

Clang can be used to compile OpenCL kernels for execution on a device (e.g. GPU). It is possible to compile the kernel into a binary (e.g. for AMDGPU) that can be uploaded to run directly on a device (e.g. using clCreateProgramWithBinary) or into generic bitcode files loadable into other toolchains.

Compiling to a binary using the default target from the installation can be done as follows:

```
$ echo "kernel void k(){}" > test.cl
$ clang test.cl
```

Compiling for a specific target can be done by specifying the triple corresponding to the target, for example:

```
$ clang -target nvptx64-unknown-unknown test.cl
$ clang -target amdgcn-amd-amdhsa -mcpu=gfx900 test.cl
```

Compiling to bitcode can be done as follows:

```
$ clang -c -emit-llvm test.cl
```

This will produce a file *test.bc* that can be used in vendor toolchains to perform machine code generation.

Note that if compiled to bitcode for generic targets such as SPIR/SPIR-V, portable IR is produced that can be used with various vendor tools as well as open source tools such as SPIRV-LLVM Translator to produce SPIR-V binary. More details are provided in the offline compilation from OpenCL kernel sources into SPIR-V using open source tools. From clang 14 onwards SPIR-V can be generated directly as detailed in the SPIR-V support section.

Clang currently supports OpenCL C language standards up to v2.0. Clang mainly supports full profile. There is only very limited support of the embedded profile. From clang 9 a C++ mode is available for OpenCL (see C++ for OpenCL).

OpenCL v3.0 support is complete but it remains in experimental state, see more details about the experimental features and limitations in OpenCL Support page.

OpenCL Specific Options

Most of the OpenCL build options from the specification v2.0 section 5.8.4 are available.

Examples:

```
$ clang -cl-std=CL2.0 -cl-single-precision-constant test.cl
```

Many flags used for the compilation for C sources can also be passed while compiling for OpenCL, examples: `-c`, `-O<1-4|s>`, `-o`, `-emit-llvm`, etc.

Some extra options are available to support special OpenCL features.

`-cl-no-stdinc`

Allows to disable all extra types and functions that are not native to the compiler. This might reduce the compilation speed marginally but many declarations from the OpenCL standard will not be accessible. For example, the following will fail to compile.

```
$ echo "bool is_wg_uniform(int i){return get_enqueued_local_size(i)==get_local_size(i);}" > test.cl
$ clang -cl-std=CL2.0 -cl-no-stdinc test.cl
error: use of undeclared identifier 'get_enqueued_local_size'
error: use of undeclared identifier 'get_local_size'
```

More information about the standard types and functions is provided in the section on the OpenCL Header.

`-cl-ext`

Enables/Disables support of OpenCL extensions and optional features. All OpenCL targets set a list of extensions that they support. Clang allows to amend this using the `-cl-ext` flag with a comma-separated list of extensions prefixed with `'+'` or `'-'`. The syntax: `-cl-ext=<(['-'|'+']<extension>[,])+>`, where extensions can be either one of the OpenCL published extensions or any vendor extension. Alternatively, `'all'` can be used to enable or disable all known extensions.

Example disabling double support for the 64-bit SPIR-V target:

```
$ clang -c -target spirv64 -cl-ext=-cl_khr_fp64 test.cl
```

Enabling all extensions except double support in R600 AMD GPU can be done using:

```
$ clang -target r600 -cl-ext=-all,+cl_khr_fp16 test.cl
```

Note that some generic targets e.g. SPIR/SPIR-V enable all extensions/features in clang by default.

OpenCL Targets

OpenCL targets are derived from the regular Clang target classes. The OpenCL specific parts of the target representation provide address space mapping as well as a set of supported extensions.

Specific Targets

There is a set of concrete HW architectures that OpenCL can be compiled for.

- For AMD target:

  ```
  $ clang -target amdgcn-amd-amdhsa -mcpu=gfx900 test.cl
  ```

- For Nvidia architectures:

  ```
  $ clang -target nvptx64-unknown-unknown test.cl
  ```

Generic Targets

- A SPIR-V binary can be produced for 32 or 64 bit targets.

  ```
  $ clang -target spirv32 -c test.cl
  $ clang -target spirv64 -c test.cl
  ```

  More details can be found in the SPIR-V support section.

- SPIR is available as a generic target to allow portable bitcode to be produced that can be used across GPU toolchains. The implementation follows the SPIR specification. There are two flavors available for 32 and 64 bits.

  ```
  $ clang -target spir test.cl -emit-llvm -c
  $ clang -target spir64 test.cl -emit-llvm -c
  ```

  Clang will generate SPIR v1.2 compatible IR for OpenCL versions up to 2.0 and SPIR v2.0 for OpenCL v2.0 or C++ for OpenCL.

- x86 is used by some implementations that are x86 compatible and currently remains for backwards compatibility (with older implementations prior to SPIR target support). For “non-SPMD” targets which cannot spawn multiple work-items on the fly using hardware, which covers practically all non-GPU devices such as

CPUs and DSPs, additional processing is needed for the kernels to support multiple work-item execution. For this, a 3rd party toolchain, such as for example POCL, can be used.

This target does not support multiple memory segments and, therefore, the fake address space map can be added using the -ffake-address-space-map flag.

All known OpenCL extensions and features are set to supported in the generic targets, however **`-cl-ext`** flag can be used to toggle individual extensions and features.

OpenCL Header

By default Clang will include standard headers and therefore most of OpenCL builtin functions and types are available during compilation. The default declarations of non-native compiler types and functions can be disabled by using flag -cl-no-stdinc.

The following example demonstrates that OpenCL kernel sources with various standard builtin functions can be compiled without the need for an explicit includes or compiler flags.

```
$ echo "bool is_wg_uniform(int i){return get_enqueued_local_size(i)==get_local_size(i);}" > test.cl
$ clang -cl-std=CL2.0 test.cl
```

More information about the default headers is provided in OpenCL Support.

OpenCL Extensions

Most of the `cl_khr_*` extensions to OpenCL C from the official OpenCL registry are available and configured per target depending on the support available in the specific architecture.

It is possible to alter the default extensions setting per target using `-cl-ext` flag. (See flags description for more details).

Vendor extensions can be added flexibly by declaring the list of types and functions associated with each extensions enclosed within the following compiler pragma directives:

```
#pragma OPENCL EXTENSION the_new_extension_name : begin
// declare types and functions associated with the extension here
#pragma OPENCL EXTENSION the_new_extension_name : end
```

For example, parsing the following code adds `my_t` type and `my_func` function to the custom `my_ext` extension.

```
#pragma OPENCL EXTENSION my_ext : begin
typedef struct{
  int a;
}my_t;
void my_func(my_t);
#pragma OPENCL EXTENSION my_ext : end
```

There is no conflict resolution for identifier clashes among extensions. It is therefore recommended that the identifiers are prefixed with a double underscore to avoid clashing with user space identifiers. Vendor extension should use reserved identifier prefix e.g. amd, arm, intel.

Clang also supports language extensions documented in The OpenCL C Language Extensions Documentation.

OpenCL-Specific Attributes

OpenCL support in Clang contains a set of attribute taken directly from the specification as well as additional attributes.

See also Attributes in Clang.

nosvm

Clang supports this attribute to comply to OpenCL v2.0 conformance, but it does not have any effect on the IR. For more details reffer to the specification section 6.7.2

opencl_unroll_hint

The implementation of this feature mirrors the unroll hint for C. More details on the syntax can be found in the specification section 6.11.5

convergent

To make sure no invalid optimizations occur for single program multiple data (SPMD) / single instruction multiple thread (SIMT) Clang provides attributes that can be used for special functions that have cross work item semantics. An example is the subgroup operations such as intel_sub_group_shuffle

```
// Define custom my_sub_group_shuffle(data, c)
// that makes use of intel_sub_group_shuffle
r1 = ...
if (r0) r1 = computeA();
// Shuffle data from r1 into r3
// of threads id r2.
r3 = my_sub_group_shuffle(r1, r2);
if (r0) r3 = computeB();
```

with non-SPMD semantics this is optimized to the following equivalent code:

```
r1 = ...
if (!r0)
  // Incorrect functionality! The data in r1
  // have not been computed by all threads yet.
  r3 = my_sub_group_shuffle(r1, r2);
else {
  r1 = computeA();
  r3 = my_sub_group_shuffle(r1, r2);
  r3 = computeB();
}
```

Declaring the function `my_sub_group_shuffle` with the convergent attribute would prevent this:

```
my_sub_group_shuffle() __attribute__((convergent));
```

Using `convergent` guarantees correct execution by keeping CFG equivalence wrt operations marked as `convergent`. CFG `G´` is equivalent to `G` wrt node `Ni` : `iff ∀ Nj (i≠j)` domination and post-domination relations with respect to `Ni` remain the same in both `G` and `G´`.

noduplicate

`noduplicate` is more restrictive with respect to optimizations than `convergent` because a convergent function only preserves CFG equivalence. This allows some optimizations to happen as long as the control flow remains unmodified.

```
for (int i=0; i<4; i++)
  my_sub_group_shuffle()
```

can be modified to:

```
my_sub_group_shuffle();
my_sub_group_shuffle();
my_sub_group_shuffle();
my_sub_group_shuffle();
```

while using `noduplicate` would disallow this. Also `noduplicate` doesn't have the same safe semantics of CFG as `convergent` and can cause changes in CFG that modify semantics of the original program.

`noduplicate` is kept for backwards compatibility only and it considered to be deprecated for future uses.

C++ for OpenCL

Starting from clang 9 kernel code can contain C++17 features: classes, templates, function overloading, type deduction, etc. Please note that this is not an implementation of OpenCL C++ and there is no plan to support it in clang in any new releases in the near future.

Clang currently supports C++ for OpenCL 1.0 and 2021. For detailed information about this language refer to the C++ for OpenCL Programming Language Documentation available in the latest build or in the official release.

To enable the C++ for OpenCL mode, pass one of following command line options when compiling `.clcpp` file:

- C++ for OpenCL 1.0: `-cl-std=clc++`, `-cl-std=CLC++`, `-cl-std=clc++1.0`, `-cl-std=CLC++1.0`, `-std=clc++`, `-std=CLC++`, `-std=clc++1.0` or `-std=CLC++1.0`.
- C++ for OpenCL 2021: `-cl-std=clc++2021`, `-cl-std=CLC++2021`, `-std=clc++2021`, `-std=CLC++2021`.

Example of use:

```
template<class T> T add( T x, T y )
{
  return x + y;
}

__kernel void test( __global float* a, __global float* b)
{
  auto index = get_global_id(0);
  a[index] = add(b[index], b[index+1]);
}

clang -cl-std=clc++1.0 test.clcpp
clang -cl-std=clc++ -c -target spirv64 test.cl
```

By default, files with `.clcpp` extension are compiled with the C++ for OpenCL 1.0 mode.

```
clang test.clcpp
```

For backward compatibility files with `.cl` extensions can also be compiled in C++ for OpenCL mode but the desirable language mode must be activated with a flag.

```
clang -cl-std=clc++ test.cl
```

Support of C++ for OpenCL 2021 is currently in experimental phase, refer to OpenCL Support for more details.

C++ for OpenCL kernel sources can also be compiled online in drivers supporting cl_ext_cxx_for_opencl extension.

Constructing and destroying global objects

Global objects with non-trivial constructors require the constructors to be run before the first kernel using the global objects is executed. Similarly global objects with non-trivial destructors require destructor invocation just after the last kernel using the program objects is executed. In OpenCL versions earlier than v2.2 there is no support for invoking global constructors. However, an easy workaround is to manually enqueue the constructor initialization kernel that has the following name scheme `_GLOBAL__sub_I_<compiled file name>`. This kernel is only present if there are global objects with non-trivial constructors present in the compiled binary. One way to check this is by passing `CL_PROGRAM_KERNEL_NAMES` to `clGetProgramInfo` (OpenCL v2.0 s5.8.7) and then checking whether any kernel name matches the naming scheme of global constructor initialization kernel above.

Note that if multiple files are compiled and linked into libraries, multiple kernels that initialize global objects for multiple modules would have to be invoked.

Applications are currently required to run initialization of global objects manually before running any kernels in which the objects are used.

```
clang -cl-std=clc++ test.cl
```

If there are any global objects to be initialized, the final binary will contain the `_GLOBAL__sub_I_test.cl` kernel to be enqueued.

Note that the manual workaround only applies to objects declared at the program scope. There is no manual workaround for the construction of static objects with non-trivial constructors inside functions.

Global destructors can not be invoked manually in the OpenCL v2.0 drivers. However, all memory used for program scope objects should be released on `clReleaseProgram`.

Libraries

Limited experimental support of C++ standard libraries for OpenCL is described in OpenCL Support page.

Target-Specific Features and Limitations

CPU Architectures Features and Limitations

X86

The support for X86 (both 32-bit and 64-bit) is considered stable on Darwin (macOS), Linux, FreeBSD, and Dragonfly BSD: it has been tested to correctly compile many large C, C++, Objective-C, and Objective-C++ codebases.

On `x86_64-mingw32`, passing i128(by value) is incompatible with the Microsoft x64 calling convention. You might need to tweak `WinX86_64ABIInfo::classify()` in lib/CodeGen/TargetInfo.cpp.

For the X86 target, clang supports the *-m16* command line argument which enables 16-bit code output. This is broadly similar to using `asm(".code16gcc")` with the GNU toolchain. The generated code and the ABI remains 32-bit but the assembler emits instructions appropriate for a CPU running in 16-bit mode, with address-size and operand-size prefixes to enable 32-bit addressing and operations.

Several micro-architecture levels as specified by the x86-64 psABI are defined. They are cumulative in the sense that features from previous levels are implicitly included in later levels.

- `-march=x86-64`: CMOV, CMPXCHG8B, FPU, FXSR, MMX, FXSR, SCE, SSE, SSE2
- `-march=x86-64-v2`: (close to Nehalem) CMPXCHG16B, LAHF-SAHF, POPCNT, SSE3, SSE4.1, SSE4.2, SSSE3
- `-march=x86-64-v3`: (close to Haswell) AVX, AVX2, BMI1, BMI2, F16C, FMA, LZCNT, MOVBE, XSAVE
- `-march=x86-64-v4`: AVX512F, AVX512BW, AVX512CD, AVX512DQ, AVX512VL

ARM

The support for ARM (specifically ARMv6 and ARMv7) is considered stable on Darwin (iOS): it has been tested to correctly compile many large C, C++, Objective-C, and Objective-C++ codebases. Clang only supports a limited number of ARM architectures. It does not yet fully support ARMv5, for example.

PowerPC

The support for PowerPC (especially PowerPC64) is considered stable on Linux and FreeBSD: it has been tested to correctly compile many large C and C++ codebases. PowerPC (32bit) is still missing certain features (e.g. PIC code on ELF platforms).

Other platforms

clang currently contains some support for other architectures (e.g. Sparc); however, significant pieces of code generation are still missing, and they haven't undergone significant testing.

clang contains limited support for the MSP430 embedded processor, but both the clang support and the LLVM backend support are highly experimental.

Other platforms are completely unsupported at the moment. Adding the minimal support needed for parsing and semantic analysis on a new platform is quite easy; see `lib/Basic/Targets.cpp` in the clang source tree. This level of support is also sufficient for conversion to LLVM IR for simple programs. Proper support for conversion to LLVM IR requires adding code to `lib/CodeGen/CGCall.cpp` at the moment; this is likely to change soon, though. Generating assembly requires a suitable LLVM backend.

Operating System Features and Limitations

Windows

Clang has experimental support for targeting "Cygming" (Cygwin / MinGW) platforms.

See also Microsoft Extensions.

Cygwin

Clang works on Cygwin-1.7.

MinGW32

Clang works on some mingw32 distributions. Clang assumes directories as below;

- `C:/mingw/include`
- `C:/mingw/lib`
- `C:/mingw/lib/gcc/mingw32/4.[3-5].0/include/c++`

On MSYS, a few tests might fail.

MinGW-w64

For 32-bit (i686-w64-mingw32), and 64-bit (x86_64-w64-mingw32), Clang assumes as below;

- `GCC versions 4.5.0 to 4.5.3, 4.6.0 to 4.6.2, or 4.7.0 (for the C++ header search path)`
  - `some_directory/bin/gcc.exe`
  - `some_directory/bin/clang.exe`
  - `some_directory/bin/clang++.exe`
  - `some_directory/bin/../include/c++/GCC_version`
  - `some_directory/bin/../include/c++/GCC_version/x86_64-w64-mingw32`
  - `some_directory/bin/../include/c++/GCC_version/i686-w64-mingw32`
  - `some_directory/bin/../include/c++/GCC_version/backward`
  - `some_directory/bin/../x86_64-w64-mingw32/include`
  - `some_directory/bin/../i686-w64-mingw32/include`
  - `some_directory/bin/../include`

This directory layout is standard for any toolchain you will find on the official MinGW-w64 website.

Clang expects the GCC executable "gcc.exe" compiled for `i686-w64-mingw32` (or `x86_64-w64-mingw32`) to be present on PATH.

Some tests might fail on `x86_64-w64-mingw32`.

AIX

The `-mdefault-visibility-export-mapping=` option can be used to control mapping of default visibility to an explicit shared object export (i.e. XCOFF exported visibility). Three values are provided for the option:

- `-mdefault-visibility-export-mapping=none`: no additional export information is created for entities with default visibility.

- `-mdefault-visibility-export-mapping=explicit`: mark entities for export if they have explict (e.g. via an attribute) default visibility from the source, including RTTI.
- `-mdefault-visibility-export-mapping=all`: set XCOFF exported visibility for all entities with default visibility from any source. This gives a export behavior similar to ELF platforms where all entities with default visibility are exported.

SPIR-V support

Clang supports generation of SPIR-V conformant to the OpenCL Environment Specification.

To generate SPIR-V binaries, Clang uses the external `llvm-spirv` tool from the SPIRV-LLVM-Translator repo.

Prior to the generation of SPIR-V binary with Clang, `llvm-spirv` should be built or installed. Please refer to the following instructions for more details. Clang will expect the `llvm-spirv` executable to be present in the `PATH` environment variable. Clang uses `llvm-spirv` with the widely adopted assembly syntax package.

The versioning of `llvm-spirv` is aligned with Clang major releases. The same applies to the main development branch. It is therefore important to ensure the `llvm-spirv` version is in alignment with the Clang version. For troubleshooting purposes `llvm-spirv` can be tested in isolation.

Example usage for OpenCL kernel compilation:

```
$ clang -target spirv32 -c test.cl
$ clang -target spirv64 -c test.cl
```

Both invocations of Clang will result in the generation of a SPIR-V binary file *test.o* for 32 bit and 64 bit respectively. This file can be imported by an OpenCL driver that support SPIR-V consumption or it can be compiled further by offline SPIR-V consumer tools.

Converting to SPIR-V produced with the optimization levels other than *-O0* is currently available as an experimental feature and it is not guaranteed to work in all cases.

Clang also supports integrated generation of SPIR-V without use of `llvm-spirv` tool as an experimental feature when `-fintegrated-objemitter` flag is passed in the command line.

```
$ clang -target spirv32 -fintegrated-objemitter -c test.cl
```

Note that only very basic functionality is supported at this point and therefore it is not suitable for arbitrary use cases. This feature is only enabled when clang build is configured with `-DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD=SPIRV` option.

Linking is done using `spirv-link` from the SPIRV-Tools project. Similar to other external linkers, Clang will expect `spirv-link` to be installed separately and to be present in the `PATH` environment variable. Please refer to the build and installation instructions.

```
$ clang -target spirv64 test1.cl test2.cl
```

More information about the SPIR-V target settings and supported versions of SPIR-V format can be found in the SPIR-V target guide.

clang-cl

clang-cl is an alternative command-line interface to Clang, designed for compatibility with the Visual C++ compiler, cl.exe.

To enable clang-cl to find system headers, libraries, and the linker when run from the command-line, it should be executed inside a Visual Studio Native Tools Command Prompt or a regular Command Prompt where the environment has been set up using e.g. vcvarsall.bat.

clang-cl can also be used from inside Visual Studio by selecting the LLVM Platform Toolset. The toolset is not part of the installer, but may be installed separately from the Visual Studio Marketplace. To use the toolset, select a project in Solution Explorer, open its Property Page (Alt+F7), and in the “General” section of “Configuration Properties” change “Platform Toolset” to LLVM. Doing so enables an additional Property Page for selecting the clang-cl executable to use for builds.

To use the toolset with MSBuild directly, invoke it with e.g. `/p:PlatformToolset=LLVM`. This allows trying out the clang-cl toolchain without modifying your project files.

It's also possible to point MSBuild at clang-cl without changing toolset by passing `/p:CLToolPath=c:\llvm\bin /p:CLToolExe=clang-cl.exe`.

When using CMake and the Visual Studio generators, the toolset can be set with the `-T` flag:

```
cmake -G"Visual Studio 16 2019" -T LLVM ..
```

When using CMake with the Ninja generator, set the `CMAKE_C_COMPILER` and `CMAKE_CXX_COMPILER` variables to clang-cl:

```
cmake -GNinja -DCMAKE_C_COMPILER="c:/Program Files (x86)/LLVM/bin/clang-cl.exe"
    -DCMAKE_CXX_COMPILER="c:/Program Files (x86)/LLVM/bin/clang-cl.exe" ..
```

Command-Line Options

To be compatible with cl.exe, clang-cl supports most of the same command-line options. Those options can start with either `/` or `-`. It also supports some of Clang's core options, such as the `-W` options.

Options that are known to clang-cl, but not currently supported, are ignored with a warning. For example:

```
clang-cl.exe: warning: argument unused during compilation: '/AI'
```

To suppress warnings about unused arguments, use the `-Qunused-arguments` option.

Options that are not known to clang-cl will be ignored by default. Use the `-Werror=unknown-argument` option in order to treat them as errors. If these options are spelled with a leading `/`, they will be mistaken for a filename:

```
clang-cl.exe: error: no such file or directory: '/foobar'
```

Please file a bug for any valid cl.exe flags that clang-cl does not understand.

Execute `clang-cl /?` to see a list of supported options:

```
CL.EXE COMPATIBILITY OPTIONS:
  /?                      Display available options
  /arch:<value>           Set architecture for code generation
  /Brepro-                Emit an object file which cannot be reproduced over time
  /Brepro                 Emit an object file which can be reproduced over time
  /clang:<arg>            Pass <arg> to the clang driver
  /C                      Don't discard comments when preprocessing
  /c                      Compile only
  /d1PP                   Retain macro definitions in /E mode
  /d1reportAllClassLayout Dump record layout information
  /diagnostics:caret      Enable caret and column diagnostics (on by default)
  /diagnostics:classic    Disable column and caret diagnostics
  /diagnostics:column     Disable caret diagnostics but keep column info
  /D <macro[=value]>      Define macro
  /EH<value>              Exception handling model
  /EP                     Disable linemarker output and preprocess to stdout
  /execution-charset:<value>
                          Runtime encoding, supports only UTF-8
  /E                      Preprocess to stdout
  /FA                     Output assembly code file during compilation
  /Fa<file or directory>  Output assembly code to this file during compilation (with /FA)
  /Fe<file or directory>  Set output executable file or directory (ends in / or \)
  /FI <value>             Include file before parsing
  /Fi<file>               Set preprocess output file name (with /P)
  /Fo<file or directory>  Set output object file, or directory (ends in / or \) (with /c)
  /fp:except-
  /fp:except
  /fp:fast
```

```
/fp:precise
/fp:strict
/Fp<filename>           Set pch filename (with /Yc and /Yu)
/GA                     Assume thread-local variables are defined in the executable
/Gd                     Set __cdecl as a default calling convention
/GF-                    Disable string pooling
/GF                     Enable string pooling (default)
/GR-                    Disable emission of RTTI data
/Gregcall               Set __regcall as a default calling convention
/GR                     Enable emission of RTTI data
/Gr                     Set __fastcall as a default calling convention
/GS-                    Disable buffer security check
/GS                     Enable buffer security check (default)
/Gs                     Use stack probes (default)
/Gs<value>              Set stack probe size (default 4096)
/guard:<value>          Enable Control Flow Guard with /guard:cf,
                        or only the table with /guard:cf,nochecks.
                        Enable EH Continuation Guard with /guard:ehcont
/Gv                     Set __vectorcall as a default calling convention
/Gw-                    Don't put each data item in its own section
/Gw                     Put each data item in its own section
/GX-                    Disable exception handling
/GX                     Enable exception handling
/Gy-                    Don't put each function in its own section (default)
/Gy                     Put each function in its own section
/Gz                     Set __stdcall as a default calling convention
/help                   Display available options
/imsvc <dir>            Add directory to system include search path, as if part of %INCLUDE%
/I <dir>                Add directory to include search path
/J                      Make char type unsigned
/LDd                    Create debug DLL
/LD                     Create DLL
/link <options>         Forward options to the linker
/MDd                    Use DLL debug run-time
/MD                     Use DLL run-time
/MTd                    Use static debug run-time
/MT                     Use static run-time
/O0                     Disable optimization
/O1                     Optimize for size  (same as /Og     /Os /Oy /Ob2 /GF /Gy)
/O2                     Optimize for speed (same as /Og /Oi /Ot /Oy /Ob2 /GF /Gy)
/Ob0                    Disable function inlining
/Ob1                    Only inline functions which are (explicitly or implicitly) marked inline
/Ob2                    Inline functions as deemed beneficial by the compiler
/Od                     Disable optimization
/Og                     No effect
/Oi-                    Disable use of builtin functions
/Oi                     Enable use of builtin functions
/Os                     Optimize for size
/Ot                     Optimize for speed
/Ox                     Deprecated (same as /Og /Oi /Ot /Oy /Ob2); use /O2 instead
/Oy-                    Disable frame pointer omission (x86 only, default)
/Oy                     Enable frame pointer omission (x86 only)
/O<flags>               Set multiple /O flags at once; e.g. '/O2y-' for '/O2 /Oy-'
/o <file or directory>  Set output file or directory (ends in / or \)
/P                      Preprocess to file
/Qvec-                  Disable the loop vectorization passes
/Qvec                   Enable the loop vectorization passes
/showFilenames-         Don't print the name of each compiled file (default)
/showFilenames          Print the name of each compiled file
```

```
  /showIncludes           Print info about included files to stderr
  /source-charset:<value> Source encoding, supports only UTF-8
  /std:<value>            Language standard to compile for
  /TC                     Treat all source files as C
  /Tc <filename>          Specify a C source file
  /TP                     Treat all source files as C++
  /Tp <filename>          Specify a C++ source file
  /utf-8                  Set source and runtime encoding to UTF-8 (default)
  /U <macro>              Undefine macro
  /vd<value>              Control vtordisp placement
  /vmb                    Use a best-case representation method for member pointers
  /vmg                    Use a most-general representation for member pointers
  /vmm                    Set the default most-general representation to multiple inheritance
  /vms                    Set the default most-general representation to single inheritance
  /vmv                    Set the default most-general representation to virtual inheritance
  /volatile:iso           Volatile loads and stores have standard semantics
  /volatile:ms            Volatile loads and stores have acquire and release semantics
  /W0                     Disable all warnings
  /W1                     Enable -Wall
  /W2                     Enable -Wall
  /W3                     Enable -Wall
  /W4                     Enable -Wall and -Wextra
  /Wall                   Enable -Weverything
  /WX-                    Do not treat warnings as errors
  /WX                     Treat warnings as errors
  /w                      Disable all warnings
  /X                      Don't add %INCLUDE% to the include search path
  /Y-                     Disable precompiled headers, overrides /Yc and /Yu
  /Yc<filename>           Generate a pch file for all code up to and including <filename>
  /Yu<filename>           Load a pch file and use it instead of all code up to and including <filename>
  /Z7                     Enable CodeView debug information in object files
  /Zc:char8_t             Enable C++2a char8_t type
  /Zc:char8_t-            Disable C++2a char8_t type
  /Zc:dllexportInlines-   Don't dllexport/dllimport inline member functions of dllexport/import classes
  /Zc:dllexportInlines    dllexport/dllimport inline member functions of dllexport/import classes (default)
  /Zc:sizedDealloc-       Disable C++14 sized global deallocation functions
  /Zc:sizedDealloc        Enable C++14 sized global deallocation functions
  /Zc:strictStrings       Treat string literals as const
  /Zc:threadSafeInit-     Disable thread-safe initialization of static variables
  /Zc:threadSafeInit      Enable thread-safe initialization of static variables
  /Zc:trigraphs-          Disable trigraphs (default)
  /Zc:trigraphs           Enable trigraphs
  /Zc:twoPhase-           Disable two-phase name lookup in templates
  /Zc:twoPhase            Enable two-phase name lookup in templates
  /Zi                     Alias for /Z7. Does not produce PDBs.
  /Zl                     Don't mention any default libraries in the object file
  /Zp                     Set the default maximum struct packing alignment to 1
  /Zp<value>              Specify the default maximum struct packing alignment
  /Zs                     Syntax-check only

OPTIONS:
  -###                    Print (but do not run) the commands to run for this compilation
  --analyze               Run the static analyzer
  -faddrsig               Emit an address-significance table
  -fansi-escape-codes     Use ANSI escape codes for diagnostics
  -fblocks                Enable the 'blocks' language feature
  -fcf-protection=<value> Instrument control-flow architecture protection. Options: return, branch, full, none.
  -fcf-protection         Enable cf-protection in 'full' mode
  -fcolor-diagnostics     Use colors in diagnostics
```

```
-fcomplete-member-pointers
                      Require member pointer base types to be complete if they would be significant under the Microsoft ABI
-fcoverage-mapping    Generate coverage mapping to enable code coverage analysis
-fcrash-diagnostics-dir=<dir>
                      Put crash-report files in <dir>
-fdebug-macro         Emit macro debug information
-fdelayed-template-parsing
                      Parse templated function definitions at the end of the translation unit
-fdiagnostics-absolute-paths
                      Print absolute paths in diagnostics
-fdiagnostics-parseable-fixits
                      Print fix-its in machine parseable form
-flto=<value>         Set LTO mode to either 'full' or 'thin'
-flto                 Enable LTO in 'full' mode
-fmerge-all-constants Allow merging of constants
-fms-compatibility-version=<value>
                      Dot-separated value representing the Microsoft compiler version
                      number to report in _MSC_VER (0 = don't define it (default))
-fms-compatibility    Enable full Microsoft Visual C++ compatibility
-fms-extensions       Accept some non-standard constructs supported by the Microsoft compiler
-fmsc-version=<value> Microsoft compiler version number to report in _MSC_VER
                      (0 = don't define it (default))
-fno-addrsig          Don't emit an address-significance table
-fno-builtin-<value>  Disable implicit builtin knowledge of a specific function
-fno-builtin          Disable implicit builtin knowledge of functions
-fno-complete-member-pointers
                      Do not require member pointer base types to be complete if they would be significant under the Microsoft ABI
-fno-coverage-mapping Disable code coverage analysis
-fno-crash-diagnostics Disable auto-generation of preprocessed source files and a script for reproduction during a clang crash
-fno-debug-macro      Do not emit macro debug information
-fno-delayed-template-parsing
                      Disable delayed template parsing
-fno-sanitize-address-poison-custom-array-cookie
                      Disable poisoning array cookies when using custom operator new[] in AddressSanitizer
-fno-sanitize-address-use-after-scope
                      Disable use-after-scope detection in AddressSanitizer
-fno-sanitize-address-use-odr-indicator
                       Disable ODR indicator globals
-fno-sanitize-ignorelist Don't use ignorelist file for sanitizers
-fno-sanitize-cfi-cross-dso
                      Disable control flow integrity (CFI) checks for cross-DSO calls.
-fno-sanitize-coverage=<value>
                      Disable specified features of coverage instrumentation for Sanitizers
-fno-sanitize-memory-track-origins
                      Disable origins tracking in MemorySanitizer
-fno-sanitize-memory-use-after-dtor
                      Disable use-after-destroy detection in MemorySanitizer
-fno-sanitize-recover=<value>
                      Disable recovery for specified sanitizers
-fno-sanitize-stats   Disable sanitizer statistics gathering.
-fno-sanitize-thread-atomics
                      Disable atomic operations instrumentation in ThreadSanitizer
-fno-sanitize-thread-func-entry-exit
                      Disable function entry/exit instrumentation in ThreadSanitizer
-fno-sanitize-thread-memory-access
                      Disable memory access instrumentation in ThreadSanitizer
-fno-sanitize-trap=<value>
                      Disable trapping for specified sanitizers
-fno-standalone-debug Limit debug information produced to reduce size of debug binary
```

```
-fobjc-runtime=<value>  Specify the target Objective-C runtime kind and version
-fprofile-exclude-files=<value>
                        Instrument only functions from files where names don't match all the regexes separated by a semi-colon
-fprofile-filter-files=<value>
                        Instrument only functions from files where names match any regex separated by a semi-colon
-fprofile-instr-generate=<file>
                        Generate instrumented code to collect execution counts into <file>
                        (overridden by LLVM_PROFILE_FILE env var)
-fprofile-instr-generate
                        Generate instrumented code to collect execution counts into default.profraw file
                        (overridden by '=' form of option or LLVM_PROFILE_FILE env var)
-fprofile-instr-use=<value>
                        Use instrumentation data for profile-guided optimization
-fprofile-remapping-file=<file>
                        Use the remappings described in <file> to match the profile data against names in the program
-fprofile-list=<file>
                        Filename defining the list of functions/files to instrument
-fsanitize-address-field-padding=<value>
                        Level of field padding for AddressSanitizer
-fsanitize-address-globals-dead-stripping
                        Enable linker dead stripping of globals in AddressSanitizer
-fsanitize-address-poison-custom-array-cookie
                        Enable poisoning array cookies when using custom operator new[] in AddressSanitizer
-fsanitize-address-use-after-return=<mode>
                        Select the mode of detecting stack use-after-return in AddressSanitizer: never | runtime (default) | always
-fsanitize-address-use-after-scope
                        Enable use-after-scope detection in AddressSanitizer
-fsanitize-address-use-odr-indicator
                        Enable ODR indicator globals to avoid false ODR violation reports in partially sanitized programs at the cost of an increase in binary size
-fsanitize-ignorelist=<value>
                        Path to ignorelist file for sanitizers
-fsanitize-cfi-cross-dso
                        Enable control flow integrity (CFI) checks for cross-DSO calls.
-fsanitize-cfi-icall-generalize-pointers
                        Generalize pointers in CFI indirect call type signature checks
-fsanitize-coverage=<value>
                        Specify the type of coverage instrumentation for Sanitizers
-fsanitize-hwaddress-abi=<value>
                        Select the HWAddressSanitizer ABI to target (interceptor or platform, default interceptor)
-fsanitize-memory-track-origins=<value>
                        Enable origins tracking in MemorySanitizer
-fsanitize-memory-track-origins
                        Enable origins tracking in MemorySanitizer
-fsanitize-memory-use-after-dtor
                        Enable use-after-destroy detection in MemorySanitizer
-fsanitize-recover=<value>
                        Enable recovery for specified sanitizers
-fsanitize-stats        Enable sanitizer statistics gathering.
-fsanitize-thread-atomics
                        Enable atomic operations instrumentation in ThreadSanitizer (default)
-fsanitize-thread-func-entry-exit
                        Enable function entry/exit instrumentation in ThreadSanitizer (default)
-fsanitize-thread-memory-access
                        Enable memory access instrumentation in ThreadSanitizer (default)
-fsanitize-trap=<value> Enable trapping for specified sanitizers
-fsanitize-undefined-strip-path-components=<number>
                        Strip (or keep only, if negative) a given number of path components when emitting check metadata.
-fsanitize=<check>      Turn on runtime checks for various forms of undefined or suspicious
                        behavior. See user manual for available checks
-fsplit-lto-unit        Enables splitting of the LTO unit.
-fstandalone-debug      Emit full debug info for all types used by the program
-fwhole-program-vtables Enables whole-program vtable optimization. Requires -flto
-gcodeview-ghash        Emit type record hashes in a .debug$H section
-gcodeview              Generate CodeView debug information
-gline-directives-only  Emit debug line info directives only
-gline-tables-only      Emit debug line number tables only
-miamcu                 Use Intel MCU ABI
-mllvm <value>          Additional arguments to forward to LLVM's option processing
-nobuiltininc           Disable builtin #include directories
-Qunused-arguments      Don't emit warning for unused driver arguments
-R<remark>              Enable the specified remark
--target=<value>        Generate code for the given target
--version               Print version information
-v                      Show commands to run and use verbose output
-W<warning>             Enable the specified warning
-Xclang <arg>           Pass <arg> to the clang compiler
```

The /clang: Option

When clang-cl is run with a set of `/clang:<arg>` options, it will gather all of the `<arg>` arguments and process them as if they were passed to the clang driver. This mechanism allows you to pass flags that are not exposed in the clang-cl options or flags that have a different meaning when passed to the clang driver. Regardless of where they appear in the command line, the `/clang:` arguments are treated as if they were passed at the end of the clang-cl command line.

The /Zc:dllexportInlines- Option

This causes the class-level *dllexport* and *dllimport* attributes to not apply to inline member functions, as they otherwise would. For example, in the code below *S::foo()* would normally be defined and exported by the DLL, but when using the `/Zc:dllexportInlines-` flag it is not:

```
struct __declspec(dllexport) S {
  void foo() {}
}
```

This has the benefit that the compiler doesn't need to emit a definition of *S::foo()* in every translation unit where the declaration is included, as it would otherwise do to ensure there's a definition in the DLL even if it's not used there. If the declaration occurs in a header file that's widely used, this can save significant compilation time and output size. It also reduces the number of functions exported by the DLL similarly to what `-fvisibility-inlines-hidden`

does for shared objects on ELF and Mach-O. Since the function declaration comes with an inline definition, users of the library can use that definition directly instead of importing it from the DLL.

Note that the Microsoft Visual C++ compiler does not support this option, and if code in a DLL is compiled with `/Zc:dllexportInlines-`, the code using the DLL must be compiled in the same way so that it doesn't attempt to dllimport the inline member functions. The reverse scenario should generally work though: a DLL compiled without this flag (such as a system library compiled with Visual C++) can be referenced from code compiled using the flag, meaning that the referencing code will use the inline definitions instead of importing them from the DLL.

Also note that like when using `-fvisibility-inlines-hidden`, the address of *S::foo()* will be different inside and outside the DLL, breaking the C/C++ standard requirement that functions have a unique address.

The flag does not apply to explicit class template instantiation definitions or declarations, as those are typically used to explicitly provide a single definition in a DLL, (dllexported instantiation definition) or to signal that the definition is available elsewhere (dllimport instantiation declaration). It also doesn't apply to inline members with static local variables, to ensure that the same instance of the variable is used inside and outside the DLL.

Using this flag can cause problems when inline functions that would otherwise be dllexported refer to internal symbols of a DLL. For example:

```
void internal();

struct __declspec(dllimport) S {
  void foo() { internal(); }
}
```

Normally, references to *S::foo()* would use the definition in the DLL from which it was exported, and which presumably also has the definition of *internal()*. However, when using `/Zc:dllexportInlines-`, the inline definition of *S::foo()* is used directly, resulting in a link error since *internal()* is not available. Even worse, if there is an inline definition of *internal()* containing a static local variable, we will now refer to a different instance of that variable than in the DLL:

```
inline int internal() { static int x; return x++; }

struct __declspec(dllimport) S {
  int foo() { return internal(); }
}
```

This could lead to very subtle bugs. Using `-fvisibility-inlines-hidden` can lead to the same issue. To avoid it in this case, make *S::foo()* or *internal()* non-inline, or mark them *dllimport/dllexport* explicitly.

Finding Clang runtime libraries

clang-cl supports several features that require runtime library support:

- Address Sanitizer (ASan): `-fsanitize=address`
- Undefined Behavior Sanitizer (UBSan): `-fsanitize=undefined`
- Code coverage: `-fprofile-instr-generate -fcoverage-mapping`
- Profile Guided Optimization (PGO): `-fprofile-instr-generate`
- Certain math operations (int128 division) require the builtins library

In order to use these features, the user must link the right runtime libraries into their program. These libraries are distributed alongside Clang in the library resource directory. Clang searches for the resource directory by searching relative to the Clang executable. For example, if LLVM is installed in `C:\Program Files\LLVM`, then the profile runtime library will be located at the path `C:\Program Files\LLVM\lib\clang\11.0.0\lib\windows\clang_rt.profile-x86_64.lib`.

For UBSan, PGO, and coverage, Clang will emit object files that auto-link the appropriate runtime library, but the user generally needs to help the linker (whether it is `lld-link.exe` or MSVC `link.exe`) find the library resource directory. Using the example installation above, this would mean passing `/LIBPATH:C:\Program Files\LLVM\lib\clang\11.0.0\lib\windows` to the linker. If the user links the program with the `clang` or `clang-cl` drivers, the driver will pass this flag for them.

If the linker cannot find the appropriate library, it will emit an error like this:

```
$ clang-cl -c -fsanitize=undefined t.cpp

$ lld-link t.obj -dll
lld-link: error: could not open 'clang_rt.ubsan_standalone-x86_64.lib': no such file or directory
lld-link: error: could not open 'clang_rt.ubsan_standalone_cxx-x86_64.lib': no such file or directory

$ link t.obj -dll -nologo
LINK : fatal error LNK1104: cannot open file 'clang_rt.ubsan_standalone-x86_64.lib'
```

To fix the error, add the appropriate `/libpath:` flag to the link line.

For ASan, as of this writing, the user is also responsible for linking against the correct ASan libraries.

If the user is using the dynamic CRT (`/MD`), then they should add `clang_rt.asan_dynamic-x86_64.lib` to the link line as a regular input. For other architectures, replace x86_64 with the appropriate name here and below.

If the user is using the static CRT (`/MT`), then different runtimes are used to produce DLLs and EXEs. To link a DLL, pass `clang_rt.asan_dll_thunk-x86_64.lib`. To link an EXE, pass `-wholearchive:clang_rt.asan-x86_64.lib`.

Assembling a Complete Toolchain

Introduction 59

Tools 60

- Clang frontend 60
- Language frontends for other languages 60
- Assembler 60
- Linker 60

Runtime libraries 61

- Compiler runtime 61
- Atomics library 61
- Unwind library 62
- Sanitizer runtime 62
- C standard library 62
- C++ ABI library 62
- C++ standard library 63

Introduction

Clang is only one component in a complete tool chain for C family programming languages. In order to assemble a complete toolchain, additional tools and runtime libraries are required. Clang is designed to interoperate with existing tools and libraries for its target platforms, and the LLVM project provides alternatives for a number of these components.

This document describes the required and optional components in a complete toolchain, where to find them, and the supported versions and limitations of each option.

Warning

This document currently describes Clang configurations on POSIX-like operating systems with the GCC-compatible `clang` driver. When targeting Windows with the MSVC-compatible `clang-cl` driver, some of the details are different.

Tools

A complete compilation of C family programming languages typically involves the following pipeline of tools, some of which are omitted in some compilations:

- **Preprocessor**: This performs the actions of the C preprocessor: expanding #includes and #defines. The `-E` flag instructs Clang to stop after this step.
- **Parsing**: This parses and semantically analyzes the source language and builds a source-level intermediate representation (“AST”), producing a precompiled header (PCH), preamble, or precompiled module file (PCM), depending on the input. The `-precompile` flag instructs Clang to stop after this step. This is the default when the input is a header file.
- **IR generation**: This converts the source-level intermediate representation into an optimizer-specific intermediate representation (IR); for Clang, this is LLVM IR. The `-emit-llvm` flag instructs Clang to stop after this step. If combined with `-S`, Clang will produce textual LLVM IR; otherwise, it will produce LLVM IR bitcode.
- **Compiler backend**: This converts the intermediate representation into target-specific assembly code. The `-S` flag instructs Clang to stop after this step.
- **Assembler**: This converts target-specific assembly code into target-specific machine code object files. The `-c` flag instructs Clang to stop after this step.
- **Linker**: This combines multiple object files into a single image (either a shared object or an executable).

Clang provides all of these pieces other than the linker. When multiple steps are performed by the same tool, it is common for the steps to be fused together to avoid creating intermediate files.

When given an output of one of the above steps as an input, earlier steps are skipped (for instance, a `.s` file input will be assembled and linked).

The Clang driver can be invoked with the `-###` flag (this argument will need to be escaped under most shells) to see which commands it would run for the above steps, without running them. The `-v` (verbose) flag will print the commands in addition to running them.

Clang frontend

The Clang frontend (`clang -cc1`) is used to compile C family languages. The command-line interface of the frontend is considered to be an implementation detail, intentionally has no external documentation, and is subject to change without notice.

Language frontends for other languages

Clang can be provided with inputs written in non-C-family languages. In such cases, an external tool will be used to compile the input. The currently-supported languages are:

- Ada (`-x ada`, `.ad[bs]`)
- Fortran (`-x f95`, `.f`, `.f9[05]`, `.for`, `.fpp`, case-insensitive)
- Java (`-x java`)

In each case, GCC will be invoked to compile the input.

Assembler

Clang can either use LLVM’s integrated assembler or an external system-specific tool (for instance, the GNU Assembler on GNU OSes) to produce machine code from assembly. By default, Clang uses LLVM’s integrated assembler on all targets where it is supported. If you wish to use the system assembler instead, use the `-fno-integrated-as` option.

Linker

Clang can be configured to use one of several different linkers:

- GNU ld
- GNU gold

- LLVM's lld
- MSVC's link.exe

Link-time optimization is natively supported by lld, and supported via a linker plugin when using gold.

The default linker varies between targets, and can be overridden via the `-fuse-ld=<linker name>` flag.

Runtime libraries

A number of different runtime libraries are required to provide different layers of support for C family programs. Clang will implicitly link an appropriate implementation of each runtime library, selected based on target defaults or explicitly selected by the `--rtlib=` and `--stdlib=` flags.

The set of implicitly-linked libraries depend on the language mode. As a consequence, you should use `clang++` when linking C++ programs in order to ensure the C++ runtimes are provided.

Note

There may exist other implementations for these components not described below. Please let us know how well those other implementations work with Clang so they can be added to this list!

Compiler runtime

The compiler runtime library provides definitions of functions implicitly invoked by the compiler to support operations not natively supported by the underlying hardware (for instance, 128-bit integer multiplications), and where inline expansion of the operation is deemed unsuitable.

The default runtime library is target-specific. For targets where GCC is the dominant compiler, Clang currently defaults to using libgcc_s. On most other targets, compiler-rt is used by default.

compiler-rt (LLVM)

LLVM's compiler runtime library provides a complete set of runtime library functions containing all functions that Clang will implicitly call, in `libclang_rt.builtins.<arch>.a`.

You can instruct Clang to use compiler-rt with the `--rtlib=compiler-rt` flag. This is not supported on every platform.

If using libc++ and/or libc++abi, you may need to configure them to use compiler-rt rather than libgcc_s by passing `-DLIBCXX_USE_COMPILER_RT=YES` and/or `-DLIBCXXABI_USE_COMPILER_RT=YES` to `cmake`. Otherwise, you may end up with both runtime libraries linked into your program (this is typically harmless, but wasteful).

libgcc_s (GNU)

GCC's runtime library can be used in place of compiler-rt. However, it lacks several functions that LLVM may emit references to, particularly when using Clang's `__builtin_*_overflow` family of intrinsics.

You can instruct Clang to use libgcc_s with the `--rtlib=libgcc` flag. This is not supported on every platform.

Atomics library

If your program makes use of atomic operations and the compiler is not able to lower them all directly to machine instructions (because there either is no known suitable machine instruction or the operand is not known to be suitably aligned), a call to a runtime library `__atomic_*` function will be generated. A runtime library containing these atomics functions is necessary for such programs.

compiler-rt (LLVM)

compiler-rt contains an implementation of an atomics library.

libatomic (GNU)

libgcc_s does not provide an implementation of an atomics library. Instead, GCC's libatomic library can be used to supply these when using libgcc_s.

> **Note**
>
> Clang does not currently automatically link against libatomic when using libgcc_s. You may need to manually add `-latomic` to support this configuration when using non-native atomic operations (if you see link errors referring to `__atomic_*` functions).

Unwind library

The unwind library provides a family of `_Unwind_*` functions implementing the language-neutral stack unwinding portion of the Itanium C++ ABI (Level I). It is a dependency of the C++ ABI library, and sometimes is a dependency of other runtimes.

libunwind (LLVM)

LLVM's unwinder library is part of the llvm-project git repository. To build it, pass `-DLLVM_ENABLE_RUNTIMES=libunwind` to the cmake invocation.

If using libc++abi, you may need to configure it to use libunwind rather than libgcc_s by passing `-DLIBCXXABI_USE_LLVM_UNWINDER=YES` to `cmake`. If libc++abi is configured to use some version of libunwind, that library will be implicitly linked into binaries that link to libc++abi.

libgcc_s (GNU)

libgcc_s has an integrated unwinder, and does not need an external unwind library to be provided.

libunwind (nongnu.org)

This is another implementation of the libunwind specification. See libunwind (nongnu.org).

libunwind (PathScale)

This is another implementation of the libunwind specification. See libunwind (pathscale).

Sanitizer runtime

The instrumentation added by Clang's sanitizers (`-fsanitize=...`) implicitly makes calls to a runtime library, in order to maintain side state about the execution of the program and to issue diagnostic messages when a problem is detected.

The only supported implementation of these runtimes is provided by LLVM's compiler-rt, and the relevant portion of that library (`libclang_rt.<sanitizer>.<arch>.a`) will be implicitly linked when linking with a `-fsanitize=...` flag.

C standard library

Clang supports a wide variety of C standard library implementations.

C++ ABI library

The C++ ABI library provides an implementation of the library portion of the Itanium C++ ABI, covering both the support functionality in the main Itanium C++ ABI document and Level II of the exception handling support. References to the functions and objects in this library are implicitly generated by Clang when compiling C++ code.

While it is possible to link C++ code using libstdc++ and code using libc++ together into the same program (so long as you do not attempt to pass C++ standard library objects across the boundary), it is not generally possible to have more than one C++ ABI library in a program.

The version of the C++ ABI library used by Clang will be the one that the chosen C++ standard library was linked against. Several implementations are available:

libc++abi (LLVM)

libc++abi is LLVM's implementation of this specification.

libsupc++ (GNU)

libsupc++ is GCC's implementation of this specification. However, this library is only used when libstdc++ is linked statically. The dynamic library version of libstdc++ contains a copy of libsupc++.

> **Note**
>
> Clang does not currently automatically link against libsupc++ when statically linking libstdc++. You may need to manually add `-lsupc++` to support this configuration when using `-static` or `-static-libstdc++`.

libcxxrt (PathScale)

This is another implementation of the Itanium C++ ABI specification. See libcxxrt.

C++ standard library

Clang supports use of either LLVM's libc++ or GCC's libstdc++ implementation of the C++ standard library.

libc++ (LLVM)

libc++ is LLVM's implementation of the C++ standard library, aimed at being a complete implementation of the C++ standards from C++11 onwards.

You can instruct Clang to use libc++ with the `-stdlib=libc++` flag.

libstdc++ (GNU)

libstdc++ is GCC's implementation of the C++ standard library. Clang supports libstdc++ 4.8.3 (released 2014-05-22) and later. Historically Clang implemented workarounds for issues discovered in libstdc++, and these are removed as fixed libstdc++ becomes sufficiently old.

You can instruct Clang to use libstdc++ with the `-stdlib=libstdc++` flag.

Clang Language Extensions

Introduction 124
Feature Checking Macros 124
Include File Checking Macros 127
Builtin Macros 128
Vectors and Extended Vectors 129
Matrix Types 133
Half-Precision Floating Point 134
Messages on `deprecated` and `unavailable` Attributes 135
Attributes on Enumerators 135
C++11 Attributes on using-declarations 135
'User-Specified' System Frameworks 135
Checks for Standard Language Features 136
Type Trait Primitives 141
Blocks 144
ASM Goto with Output Constraints 144
Objective-C Features 145
Initializer lists for complex numbers in C 149
OpenCL Features 150
Builtin Functions 152
Non-standard C++11 Attributes 169
Target-Specific Extensions 169
Extensions for Static Analysis 170
Extensions for Dynamic Analysis 170
Extensions for selectively disabling optimization 171
Extensions for loop hint optimizations 172
Extensions to specify floating-point flags 174
Specifying an attribute for multiple declarations (#pragma clang attribute) 176
Specifying section names for global objects (#pragma clang section) 178
Specifying Linker Options on ELF Targets 179
Evaluating Object Size Dynamically 179
Deprecating Macros 179
Restricted Expansion Macros 179
Final Macros 180
Line Control 180
Extended Integer Types 181
Intrinsics Support within Constant Expressions 181

Objective-C Literals

Introduction

Three new features were introduced into clang at the same time: *NSNumber Literals* provide a syntax for creating `NSNumber` from scalar literal expressions; *Collection Literals* provide a short-hand for creating arrays and dictionaries; *Object Subscripting* provides a way to use subscripting with Objective-C objects. Users of Apple compiler releases can use these features starting with the Apple LLVM Compiler 4.0. Users of open-source LLVM.org compiler releases can use these features starting with clang v3.1.

These language additions simplify common Objective-C programming patterns, make programs more concise, and improve the safety of container creation.

This document describes how the features are implemented in clang, and how to use them in your own programs.

NSNumber Literals

The framework class `NSNumber` is used to wrap scalar values inside objects: signed and unsigned integers (`char`, `short`, `int`, `long`, `long long`), floating point numbers (`float`, `double`), and boolean values (`BOOL`, C++ `bool`). Scalar values wrapped in objects are also known as *boxed* values.

In Objective-C, any character, numeric or boolean literal prefixed with the `'@'` character will evaluate to a pointer to an `NSNumber` object initialized with that value. C's type suffixes may be used to control the size of numeric literals.

Examples

The following program illustrates the rules for `NSNumber` literals:

```
void main(int argc, const char *argv[]) {
  // character literals.
  NSNumber *theLetterZ = @'Z';          // equivalent to [NSNumber numberWithChar:'Z']

  // integral literals.
  NSNumber *fortyTwo = @42;             // equivalent to [NSNumber numberWithInt:42]
  NSNumber *fortyTwoUnsigned = @42U;    // equivalent to [NSNumber numberWithUnsignedInt:42U]
  NSNumber *fortyTwoLong = @42L;        // equivalent to [NSNumber numberWithLong:42L]
  NSNumber *fortyTwoLongLong = @42LL;   // equivalent to [NSNumber numberWithLongLong:42LL]

  // floating point literals.
  NSNumber *piFloat = @3.141592654F;    // equivalent to [NSNumber numberWithFloat:3.141592654F]
  NSNumber *piDouble = @3.1415926535;   // equivalent to [NSNumber numberWithDouble:3.1415926535]

  // BOOL literals.
  NSNumber *yesNumber = @YES;           // equivalent to [NSNumber numberWithBool:YES]
  NSNumber *noNumber = @NO;             // equivalent to [NSNumber numberWithBool:NO]

#ifdef __cplusplus
  NSNumber *trueNumber = @true;         // equivalent to [NSNumber numberWithBool:(BOOL)true]
  NSNumber *falseNumber = @false;       // equivalent to [NSNumber numberWithBool:(BOOL)false]
#endif
}
```

Discussion

NSNumber literals only support literal scalar values after the `'@'`. Consequently, `@INT_MAX` works, but `@INT_MIN` does not, because they are defined like this:

```
#define INT_MAX   2147483647  /* max value for an int */
#define INT_MIN   (-2147483647-1) /* min value for an int */
```

The definition of `INT_MIN` is not a simple literal, but a parenthesized expression. Parenthesized expressions are supported using the Boxed Expressions syntax, which is described in the next section.

Because `NSNumber` does not currently support wrapping `long double` values, the use of a `long double NSNumber` literal (e.g. `@123.23L`) will be rejected by the compiler.

Previously, the `BOOL` type was simply a typedef for `signed char`, and `YES` and `NO` were macros that expand to `(BOOL)1` and `(BOOL)0` respectively. To support `@YES` and `@NO` expressions, these macros are now defined using new language keywords in `<objc/objc.h>`:

```
#if __has_feature(objc_bool)
#define YES             __objc_yes
#define NO              __objc_no
#else
#define YES             ((BOOL)1)
#define NO              ((BOOL)0)
#endif
```

The compiler implicitly converts `__objc_yes` and `__objc_no` to `(BOOL)1` and `(BOOL)0`. The keywords are used to disambiguate `BOOL` and integer literals.

Objective-C++ also supports `@true` and `@false` expressions, which are equivalent to `@YES` and `@NO`.

Boxed Expressions

Objective-C provides a new syntax for boxing C expressions:

```
@( <expression> )
```

Expressions of scalar (numeric, enumerated, BOOL), C string pointer and some C structures (via NSValue) are supported:

```
// numbers.
NSNumber *smallestInt = @(-INT_MAX - 1);  // [NSNumber numberWithInt:(-INT_MAX - 1)]
NSNumber *piOverTwo = @(M_PI / 2);        // [NSNumber numberWithDouble:(M_PI / 2)]

// enumerated types.
typedef enum { Red, Green, Blue } Color;
NSNumber *favoriteColor = @(Green);       // [NSNumber numberWithInt:((int)Green)]

// strings.
NSString *path = @(getenv("PATH"));       // [NSString stringWithUTF8String:(getenv("PATH"))]
NSArray *pathComponents = [path componentsSeparatedByString:@":"];

// structs.
NSValue *center = @(view.center);         // Point p = view.center;
                                          // [NSValue valueWithBytes:&p objCType:@encode(Point)];
NSValue *frame = @(view.frame);           // Rect r = view.frame;
                                          // [NSValue valueWithBytes:&r objCType:@encode(Rect)];
```

Boxed Enums

Cocoa frameworks frequently define constant values using *enums.* Although enum values are integral, they may not be used directly as boxed literals (this avoids conflicts with future `'@'`-prefixed Objective-C keywords). Instead, an enum value must be placed inside a boxed expression. The following example demonstrates configuring an `AVAudioRecorder` using a dictionary that contains a boxed enumeration value:

```
enum {
  AVAudioQualityMin = 0,
  AVAudioQualityLow = 0x20,
  AVAudioQualityMedium = 0x40,
  AVAudioQualityHigh = 0x60,
  AVAudioQualityMax = 0x7F
};

- (AVAudioRecorder *)recordToFile:(NSURL *)fileURL {
  NSDictionary *settings = @{ AVEncoderAudioQualityKey : @(AVAudioQualityMax) };
  return [[AVAudioRecorder alloc] initWithURL:fileURL settings:settings error:NULL];
}
```

The expression `@(AVAudioQualityMax)` converts `AVAudioQualityMax` to an integer type, and boxes the value accordingly. If the enum has a fixed underlying type as in:

```
typedef enum : unsigned char { Red, Green, Blue } Color;
NSNumber *red = @(Red), *green = @(Green), *blue = @(Blue); // => [NSNumber numberWithUnsignedChar:]
```

then the fixed underlying type will be used to select the correct `NSNumber` creation method.

Boxing a value of enum type will result in a `NSNumber` pointer with a creation method according to the underlying type of the enum, which can be a fixed underlying type or a compiler-defined integer type capable of representing the values of all the members of the enumeration:

```
typedef enum : unsigned char { Red, Green, Blue } Color;
Color col = Red;
NSNumber *nsCol = @(col); // => [NSNumber numberWithUnsignedChar:]
```

Boxed C Strings

A C string literal prefixed by the `'@'` token denotes an `NSString` literal in the same way a numeric literal prefixed by the `'@'` token denotes an `NSNumber` literal. When the type of the parenthesized expression is `(char *)` or `(const char *)`, the result of the boxed expression is a pointer to an `NSString` object containing equivalent character data, which is assumed to be '\0'-terminated and UTF-8 encoded. The following example converts C-style command line arguments into `NSString` objects.

```
// Partition command line arguments into positional and option arguments.
NSMutableArray *args = [NSMutableArray new];
NSMutableDictionary *options = [NSMutableDictionary new];
while (--argc) {
    const char *arg = *++argv;
    if (strncmp(arg, "--", 2) == 0) {
        options[@(arg + 2)] = @(*++argv);   // --key value
    } else {
        [args addObject:@(arg)];            // positional argument
    }
}
```

As with all C pointers, character pointer expressions can involve arbitrary pointer arithmetic, therefore programmers must ensure that the character data is valid. Passing `NULL` as the character pointer will raise an exception at runtime. When possible, the compiler will reject `NULL` character pointers used in boxed expressions.

Boxed C Structures

Boxed expressions support construction of NSValue objects. It said that C structures can be used, the only requirement is: structure should be marked with `objc_boxable` attribute. To support older version of frameworks and/or third-party libraries you may need to add the attribute via `typedef`.

```
struct __attribute__((objc_boxable)) Point {
    // ...
};

typedef struct __attribute__((objc_boxable)) _Size {
    // ...
} Size;

typedef struct _Rect {
    // ...
} Rect;

struct Point p;
NSValue *point = @(p);          // ok
Size s;
NSValue *size = @(s);           // ok

Rect r;
```

```
NSValue *bad_rect = @(r);       // error

typedef struct __attribute__((objc_boxable)) _Rect Rect;

NSValue *good_rect = @(r);      // ok
```

Container Literals

Objective-C now supports a new expression syntax for creating immutable array and dictionary container objects.

Examples

Immutable array expression:

```
NSArray *array = @[ @"Hello", NSApp, [NSNumber numberWithInt:42] ];
```

This creates an `NSArray` with 3 elements. The comma-separated sub-expressions of an array literal can be any Objective-C object pointer typed expression.

Immutable dictionary expression:

```
NSDictionary *dictionary = @{
    @"name" : NSUserName(),
    @"date" : [NSDate date],
    @"processInfo" : [NSProcessInfo processInfo]
};
```

This creates an `NSDictionary` with 3 key/value pairs. Value sub-expressions of a dictionary literal must be Objective-C object pointer typed, as in array literals. Key sub-expressions must be of an Objective-C object pointer type that implements the `<NSCopying>` protocol.

Discussion

Neither keys nor values can have the value `nil` in containers. If the compiler can prove that a key or value is `nil` at compile time, then a warning will be emitted. Otherwise, a runtime error will occur.

Using array and dictionary literals is safer than the variadic creation forms commonly in use today. Array literal expressions expand to calls to `+[NSArray arrayWithObjects:count:]`, which validates that all objects are non-`nil`. The variadic form, `+[NSArray arrayWithObjects:]` uses `nil` as an argument list terminator, which can lead to malformed array objects. Dictionary literals are similarly created with `+[NSDictionary dictionaryWithObjects:forKeys:count:]` which validates all objects and keys, unlike `+[NSDictionary dictionaryWithObjectsAndKeys:]` which also uses a `nil` parameter as an argument list terminator.

Object Subscripting

Objective-C object pointer values can now be used with C's subscripting operator.

Examples

The following code demonstrates the use of object subscripting syntax with `NSMutableArray` and `NSMutableDictionary` objects:

```
NSMutableArray *array = ...;
NSUInteger idx = ...;
id newObject = ...;
id oldObject = array[idx];
array[idx] = newObject;         // replace oldObject with newObject

NSMutableDictionary *dictionary = ...;
NSString *key = ...;
oldObject = dictionary[key];
dictionary[key] = newObject;    // replace oldObject with newObject
```

The next section explains how subscripting expressions map to accessor methods.

Subscripting Methods

Objective-C supports two kinds of subscript expressions: *array-style* subscript expressions use integer typed subscripts; *dictionary-style* subscript expressions use Objective-C object pointer typed subscripts. Each type of subscript expression is mapped to a message send using a predefined selector. The advantage of this design is flexibility: class designers are free to introduce subscripting by declaring methods or by adopting protocols. Moreover, because the method names are selected by the type of the subscript, an object can be subscripted using both array and dictionary styles.

Array-Style Subscripting

When the subscript operand has an integral type, the expression is rewritten to use one of two different selectors, depending on whether the element is being read or written. When an expression reads an element using an integral index, as in the following example:

```
NSUInteger idx = ...;
id value = object[idx];
```

it is translated into a call to `objectAtIndexedSubscript:`

```
id value = [object objectAtIndexedSubscript:idx];
```

When an expression writes an element using an integral index:

```
object[idx] = newValue;
```

it is translated to a call to `setObject:atIndexedSubscript:`

```
[object setObject:newValue atIndexedSubscript:idx];
```

These message sends are then type-checked and performed just like explicit message sends. The method used for objectAtIndexedSubscript: must be declared with an argument of integral type and a return value of some Objective-C object pointer type. The method used for setObject:atIndexedSubscript: must be declared with its first argument having some Objective-C pointer type and its second argument having integral type.

The meaning of indexes is left up to the declaring class. The compiler will coerce the index to the appropriate argument type of the method it uses for type-checking. For an instance of `NSArray`, reading an element using an index outside the range `[0, array.count)` will raise an exception. For an instance of `NSMutableArray`, assigning to an element using an index within this range will replace that element, but assigning to an element using an index outside this range will raise an exception; no syntax is provided for inserting, appending, or removing elements for mutable arrays.

A class need not declare both methods in order to take advantage of this language feature. For example, the class `NSArray` declares only `objectAtIndexedSubscript:`, so that assignments to elements will fail to type-check; moreover, its subclass `NSMutableArray` declares `setObject:atIndexedSubscript:`.

Dictionary-Style Subscripting

When the subscript operand has an Objective-C object pointer type, the expression is rewritten to use one of two different selectors, depending on whether the element is being read from or written to. When an expression reads an element using an Objective-C object pointer subscript operand, as in the following example:

```
id key = ...;
id value = object[key];
```

it is translated into a call to the `objectForKeyedSubscript:` method:

```
id value = [object objectForKeyedSubscript:key];
```

When an expression writes an element using an Objective-C object pointer subscript:

```
object[key] = newValue;
```

it is translated to a call to `setObject:forKeyedSubscript:`

```
[object setObject:newValue forKeyedSubscript:key];
```

The behavior of `setObject:forKeyedSubscript:` is class-specific; but in general it should replace an existing value if one is already associated with a key, otherwise it should add a new value for the key. No syntax is provided for removing elements from mutable dictionaries.

Discussion

An Objective-C subscript expression occurs when the base operand of the C subscript operator has an Objective-C object pointer type. Since this potentially collides with pointer arithmetic on the value, these expressions are only supported under the modern Objective-C runtime, which categorically forbids such arithmetic.

Currently, only subscripts of integral or Objective-C object pointer type are supported. In C++, a class type can be used if it has a single conversion function to an integral or Objective-C pointer type, in which case that conversion is applied and analysis continues as appropriate. Otherwise, the expression is ill-formed.

An Objective-C object subscript expression is always an l-value. If the expression appears on the left-hand side of a simple assignment operator (=), the element is written as described below. If the expression appears on the left-hand side of a compound assignment operator (e.g. +=), the program is ill-formed, because the result of reading an element is always an Objective-C object pointer and no binary operators are legal on such pointers. If the expression appears in any other position, the element is read as described below. It is an error to take the address of a subscript expression, or (in C++) to bind a reference to it.

Programs can use object subscripting with Objective-C object pointers of type `id`. Normal dynamic message send rules apply; the compiler must see *some* declaration of the subscripting methods, and will pick the declaration seen first.

Caveats

Objects created using the literal or boxed expression syntax are not guaranteed to be uniqued by the runtime, but nor are they guaranteed to be newly-allocated. As such, the result of performing direct comparisons against the location of an object literal (using ==, `!=`, <, <=, >, or >=) is not well-defined. This is usually a simple mistake in code that intended to call the `isEqual:` method (or the `compare:` method).

This caveat applies to compile-time string literals as well. Historically, string literals (using the `@"..."` syntax) have been uniqued across translation units during linking. This is an implementation detail of the compiler and should not be relied upon. If you are using such code, please use global string constants instead (`NSString * const MyConst = @"..."`) or use `isEqual:`.

Grammar Additions

To support the new syntax described above, the Objective-C @-expression grammar has the following new productions:

```
objc-at-expression : '@' (string-literal | encode-literal | selector-literal | protocol-literal | object-literal)
                   ;

object-literal : ('+' | '-')? numeric-constant
               | character-constant
               | boolean-constant
               | array-literal
               | dictionary-literal
               ;

boolean-constant : '__objc_yes' | '__objc_no' | 'true' | 'false'  /* boolean keywords. */
                 ;

array-literal : '[' assignment-expression-list ']'
              ;

assignment-expression-list : assignment-expression (',' assignment-expression-list)?
                           | /* empty */
                           ;

dictionary-literal : '{' key-value-list '}'

                     ;

key-value-list : key-value-pair (',' key-value-list)?
               | /* empty */
```

```
                ;

 key-value-pair : assignment-expression ':' assignment-expression
                ;
```

Note: `@true` and `@false` are only supported in Objective-C++.

Availability Checks

Programs test for the new features by using clang's __has_feature checks. Here are examples of their use:

```
#if __has_feature(objc_array_literals)
    // new way.
    NSArray *elements = @[ @"H", @"He", @"O", @"C" ];
#else
    // old way (equivalent).
    id objects[] = { @"H", @"He", @"O", @"C" };
    NSArray *elements = [NSArray arrayWithObjects:objects count:4];
#endif

#if __has_feature(objc_dictionary_literals)
    // new way.
    NSDictionary *masses = @{ @"H" : @1.0078,  @"He" : @4.0026, @"O" : @15.9990, @"C" : @12.0096 };
#else
    // old way (equivalent).
    id keys[] = { @"H", @"He", @"O", @"C" };
    id values[] = { [NSNumber numberWithDouble:1.0078], [NSNumber numberWithDouble:4.0026],
                    [NSNumber numberWithDouble:15.9990], [NSNumber numberWithDouble:12.0096] };
    NSDictionary *masses = [NSDictionary dictionaryWithObjects:objects forKeys:keys count:4];
#endif

#if __has_feature(objc_subscripting)
    NSUInteger i, count = elements.count;
    for (i = 0; i < count; ++i) {
        NSString *element = elements[i];
        NSNumber *mass = masses[element];
        NSLog(@"the mass of %@ is %@", element, mass);
    }
#else
    NSUInteger i, count = [elements count];
    for (i = 0; i < count; ++i) {
        NSString *element = [elements objectAtIndex:i];
        NSNumber *mass = [masses objectForKey:element];
        NSLog(@"the mass of %@ is %@", element, mass);
    }
#endif

#if __has_attribute(objc_boxable)
    typedef struct __attribute__((objc_boxable)) _Rect Rect;
#endif

#if __has_feature(objc_boxed_nsvalue_expressions)
    CABasicAnimation animation = [CABasicAnimation animationWithKeyPath:@"position"];
    animation.fromValue = @(layer.position);
    animation.toValue = @(newPosition);
    [layer addAnimation:animation forKey:@"move"];
#else
    CABasicAnimation animation = [CABasicAnimation animationWithKeyPath:@"position"];
    animation.fromValue = [NSValue valueWithCGPoint:layer.position];
    animation.toValue = [NSValue valueWithCGPoint:newPosition];
    [layer addAnimation:animation forKey:@"move"];
#endif
```

Code can use also `__has_feature(objc_bool)` to check for the availability of numeric literals support. This checks for the new `__objc_yes / __objc_no` keywords, which enable the use of `@YES / @NO` literals.

To check whether boxed expressions are supported, use `__has_feature(objc_boxed_expressions)` feature macro.

Language Specification for Blocks

Revisions 72
Overview 72
The Block Type 72
Block Variable Declarations 72
Block Literal Expressions 73
The Invoke Operator 74
The Copy and Release Operations 74
The `__block` Storage Qualifier 74
Control Flow 74
Objective-C Extensions 74
C++ Extensions 75

Revisions

- 2008/2/25 — created
- 2008/7/28 — revised, `__block` syntax
- 2008/8/13 — revised, Block globals
- 2008/8/21 — revised, C++ elaboration
- 2008/11/1 — revised, `__weak` support
- 2009/1/12 — revised, explicit return types
- 2009/2/10 — revised, `__block` objects need retain

Overview

A new derived type is introduced to C and, by extension, Objective-C, C++, and Objective-C++

The Block Type

Like function types, the Block type is a pair consisting of a result value type and a list of parameter types very similar to a function type. Blocks are intended to be used much like functions with the key distinction being that in addition to executable code they also contain various variable bindings to automatic (stack) or managed (heap) memory.

The abstract declarator,

```
int (^)(char, float)
```

describes a reference to a Block that, when invoked, takes two parameters, the first of type char and the second of type float, and returns a value of type int. The Block referenced is of opaque data that may reside in automatic (stack) memory, global memory, or heap memory.

Block Variable Declarations

A variable with Block type is declared using function pointer style notation substituting ^ for *. The following are valid Block variable declarations:

```
void (^blockReturningVoidWithVoidArgument)(void);
int (^blockReturningIntWithIntAndCharArguments)(int, char);
void (^arrayOfTenBlocksReturningVoidWithIntArgument[10])(int);
```

Variadic ... arguments are supported. [variadic.c] A Block that takes no arguments must specify void in the argument list [voidarg.c]. An empty parameter list does not represent, as K&R provide, an unspecified argument list. Note: both gcc and clang support K&R style as a convenience.

A Block reference may be cast to a pointer of arbitrary type and vice versa. [cast.c] A Block reference may not be dereferenced via the pointer dereference operator *, and thus a Block's size may not be computed at compile time. [sizeof.c]

Block Literal Expressions

A Block literal expression produces a reference to a Block. It is introduced by the use of the ^ token as a unary operator.

```
Block_literal_expression ::=   ^ block_decl compound_statement_body
block_decl ::=
block_decl ::= parameter_list
block_decl ::= type_expression
```

where type expression is extended to allow ^ as a Block reference (pointer) where * is allowed as a function reference (pointer).

The following Block literal:

```
^ void (void) { printf("hello world\n"); }
```

produces a reference to a Block with no arguments with no return value.

The return type is optional and is inferred from the return statements. If the return statements return a value, they all must return a value of the same type. If there is no value returned the inferred type of the Block is void; otherwise it is the type of the return statement value.

If the return type is omitted and the argument list is (void), the (void) argument list may also be omitted.

So:

```
^ ( void ) { printf("hello world\n"); }
```

and:

```
^ { printf("hello world\n"); }
```

are exactly equivalent constructs for the same expression.

The type_expression extends C expression parsing to accommodate Block reference declarations as it accommodates function pointer declarations.

Given:

```
typedef int (*pointerToFunctionThatReturnsIntWithCharArg)(char);
pointerToFunctionThatReturnsIntWithCharArg functionPointer;
^ pointerToFunctionThatReturnsIntWithCharArg (float x) { return functionPointer; }
```

and:

```
^ int ((*)(float x))(char) { return functionPointer; }
```

are equivalent expressions, as is:

```
^(float x) { return functionPointer; }
```

[returnfunctionptr.c]

The compound statement body establishes a new lexical scope within that of its parent. Variables used within the scope of the compound statement are bound to the Block in the normal manner with the exception of those in automatic (stack) storage. Thus one may access functions and global variables as one would expect, as well as static local variables. [testme]

Local automatic (stack) variables referenced within the compound statement of a Block are imported and captured by the Block as const copies. The capture (binding) is performed at the time of the Block literal expression evaluation.

The compiler is not required to capture a variable if it can prove that no references to the variable will actually be evaluated. Programmers can force a variable to be captured by referencing it in a statement at the beginning of the Block, like so:

```
(void) foo;
```

This matters when capturing the variable has side-effects, as it can in Objective-C or C++.

The lifetime of variables declared in a Block is that of a function; each activation frame contains a new copy of variables declared within the local scope of the Block. Such variable declarations should be allowed anywhere [testme] rather than only when C99 parsing is requested, including for statements. [testme]

Block literal expressions may occur within Block literal expressions (nest) and all variables captured by any nested blocks are implicitly also captured in the scopes of their enclosing Blocks.

A Block literal expression may be used as the initialization value for Block variables at global or local static scope.

The Invoke Operator

Blocks are invoked using function call syntax with a list of expression parameters of types corresponding to the declaration and returning a result type also according to the declaration. Given:

```
int (^x)(char);
void (^z)(void);
int (^(*y))(char) = &x;
```

the following are all legal Block invocations:

```
x('a');
(*y)('a');
(true ? x : *y)('a')
```

The Copy and Release Operations

The compiler and runtime provide copy and release operations for Block references that create and, in matched use, release allocated storage for referenced Blocks.

The copy operation `Block_copy()` is styled as a function that takes an arbitrary Block reference and returns a Block reference of the same type. The release operation, `Block_release()`, is styled as a function that takes an arbitrary Block reference and, if dynamically matched to a Block copy operation, allows recovery of the referenced allocated memory.

The `__block` Storage Qualifier

In addition to the new Block type we also introduce a new storage qualifier, __block, for local variables. [testme: a __block declaration within a block literal] The `__block` storage qualifier is mutually exclusive to the existing local storage qualifiers auto, register, and static. [testme] Variables qualified by `__block` act as if they were in allocated storage and this storage is automatically recovered after last use of said variable. An implementation may choose an optimization where the storage is initially automatic and only “moved” to allocated (heap) storage upon a Block_copy of a referencing Block. Such variables may be mutated as normal variables are.

In the case where a `__block` variable is a Block one must assume that the `__block` variable resides in allocated storage and as such is assumed to reference a Block that is also in allocated storage (that it is the result of a `Block_copy` operation). Despite this there is no provision to do a `Block_copy` or a `Block_release` if an implementation provides initial automatic storage for Blocks. This is due to the inherent race condition of potentially several threads trying to update the shared variable and the need for synchronization around disposing of older values and copying new ones. Such synchronization is beyond the scope of this language specification.

Control Flow

The compound statement of a Block is treated much like a function body with respect to control flow in that goto, break, and continue do not escape the Block. Exceptions are treated *normally* in that when thrown they pop stack frames until a catch clause is found.

Objective-C Extensions

Objective-C extends the definition of a Block reference type to be that also of id. A variable or expression of Block type may be messaged or used as a parameter wherever an id may be. The converse is also true. Block references may thus appear as properties and are subject to the assign, retain, and copy attribute logic that is reserved for objects.

All Blocks are constructed to be Objective-C objects regardless of whether the Objective-C runtime is operational in the program or not. Blocks using automatic (stack) memory are objects and may be messaged, although they may not be assigned into `__weak` locations if garbage collection is enabled.

Within a Block literal expression within a method definition references to instance variables are also imported into the lexical scope of the compound statement. These variables are implicitly qualified as references from self, and so self is imported as a const copy. The net effect is that instance variables can be mutated.

The Block_copy operator retains all objects held in variables of automatic storage referenced within the Block expression (or form strong references if running under garbage collection). Object variables of `__block` storage type are assumed to hold normal pointers with no provision for retain and release messages.

Foundation defines (and supplies) `-copy` and `-release` methods for Blocks.

In the Objective-C and Objective-C++ languages, we allow the `__weak` specifier for `__block` variables of object type. If garbage collection is not enabled, this qualifier causes these variables to be kept without retain messages being sent. This knowingly leads to dangling pointers if the Block (or a copy) outlives the lifetime of this object.

In garbage collected environments, the `__weak` variable is set to nil when the object it references is collected, as long as the `__block` variable resides in the heap (either by default or via `Block_copy()`). The initial Apple implementation does in fact start `__block` variables on the stack and migrate them to the heap only as a result of a `Block_copy()` operation.

It is a runtime error to attempt to assign a reference to a stack-based Block into any storage marked `__weak`, including `__weak __block` variables.

C++ Extensions

Block literal expressions within functions are extended to allow const use of C++ objects, pointers, or references held in automatic storage.

As usual, within the block, references to captured variables become const-qualified, as if they were references to members of a const object. Note that this does not change the type of a variable of reference type.

For example, given a class Foo:

```
Foo foo;
Foo &fooRef = foo;
Foo *fooPtr = &foo;
```

A Block that referenced these variables would import the variables as const variations:

```
const Foo block_foo = foo;
Foo &block_fooRef = fooRef;
Foo *const block_fooPtr = fooPtr;
```

Captured variables are copied into the Block at the instant of evaluating the Block literal expression. They are also copied when calling `Block_copy()` on a Block allocated on the stack. In both cases, they are copied as if the variable were const-qualified, and it's an error if there's no such constructor.

Captured variables in Blocks on the stack are destroyed when control leaves the compound statement that contains the Block literal expression. Captured variables in Blocks on the heap are destroyed when the reference count of the Block drops to zero.

Variables declared as residing in `__block` storage may be initially allocated in the heap or may first appear on the stack and be copied to the heap as a result of a `Block_copy()` operation. When copied from the stack, `__block` variables are copied using their normal qualification (i.e. without adding const). In C++11, `__block` variables are copied as x-values if that is possible, then as l-values if not; if both fail, it's an error. The destructor for any initial stack-based version is called at the variable's normal end of scope.

References to `this`, as well as references to non-static members of any enclosing class, are evaluated by capturing `this` just like a normal variable of C pointer type.

Member variables that are Blocks may not be overloaded by the types of their arguments.

Block Implementation Specification

History 76

High Level 77

Imported Variables 78

- Imported `const` copy variables 79
- Imported `const` copy of `Block` reference 79
  - Importing `__attribute__((NSObject))` variables 80
- Imported `__block` marked variables 81
  - Layout of `__block` marked variables 81
  - Access to `__block` variables from within its lexical scope 81
  - Importing `__block` variables into `Blocks` 82
  - Importing `__attribute__((NSObject))` `__block` variables 83
  - `__block` escapes 83
  - Nesting 83

Objective C Extensions to `Blocks` 83

- Importing Objects 83
- `Blocks` as Objects 83
- `__weak` `__block` Support 84

C++ Support 85

Runtime Helper Functions 86

Copyright 87

History

- 2008/7/14 - created.
- 2008/8/21 - revised, C++.
- 2008/9/24 - add `NULL` `isa` field to `__block` storage.
- 2008/10/1 - revise block layout to use a `static` descriptor structure.
- 2008/10/6 - revise block layout to use an unsigned long int flags.
- 2008/10/28 - specify use of `_Block_object_assign` and `_Block_object_dispose` for all “Object” types in helper functions.
- 2008/10/30 - revise new layout to have invoke function in same place.
- 2008/10/30 - add `__weak` support.
- 2010/3/16 - rev for stret return, signature field.
- 2010/4/6 - improved wording.
- 2013/1/6 - improved wording and converted to rst.

This document describes the Apple ABI implementation specification of Blocks.

The first shipping version of this ABI is found in Mac OS X 10.6, and shall be referred to as 10.6.ABI. As of 2010/3/16, the following describes the ABI contract with the runtime and the compiler, and, as necessary, will be referred to as ABI.2010.3.16.

Since the Apple ABI references symbols from other elements of the system, any attempt to use this ABI on systems prior to SnowLeopard is undefined.

High Level

The ABI of `Blocks` consist of their layout and the runtime functions required by the compiler. A `Block` of type `R (^)(P...)` consists of a structure of the following form:

```
struct Block_literal_1 {
    void *isa; // initialized to &_NSConcreteStackBlock or &_NSConcreteGlobalBlock
    int flags;
    int reserved;
    R (*invoke)(struct Block_literal_1 *, P...);
    struct Block_descriptor_1 {
        unsigned long int reserved;     // NULL
        unsigned long int size;         // sizeof(struct Block_literal_1)
        // optional helper functions
        void (*copy_helper)(void *dst, void *src);     // IFF (1<<25)
        void (*dispose_helper)(void *src);             // IFF (1<<25)
        // required ABI.2010.3.16
        const char *signature;                         // IFF (1<<30)
    } *descriptor;
    // imported variables
};
```

The following flags bits are in use thusly for a possible ABI.2010.3.16:

```
enum {
    // Set to true on blocks that have captures (and thus are not true
    // global blocks) but are known not to escape for various other
    // reasons. For backward compatibility with old runtimes, whenever
    // BLOCK_IS_NOESCAPE is set, BLOCK_IS_GLOBAL is set too. Copying a
    // non-escaping block returns the original block and releasing such a
    // block is a no-op, which is exactly how global blocks are handled.
    BLOCK_IS_NOESCAPE      =  (1 << 23),

    BLOCK_HAS_COPY_DISPOSE =  (1 << 25),
    BLOCK_HAS_CTOR =          (1 << 26), // helpers have C++ code
    BLOCK_IS_GLOBAL =         (1 << 28),
    BLOCK_HAS_STRET =         (1 << 29), // IFF BLOCK_HAS_SIGNATURE
    BLOCK_HAS_SIGNATURE =     (1 << 30),
};
```

In 10.6.ABI the (1<<29) was usually set and was always ignored by the runtime - it had been a transitional marker that did not get deleted after the transition. This bit is now paired with (1<<30), and represented as the pair (3<<30), for the following combinations of valid bit settings, and their meanings:

```
switch (flags & (3<<29)) {
  case (0<<29):      10.6.ABI, no signature field available
  case (1<<29):      10.6.ABI, no signature field available
  case (2<<29): ABI.2010.3.16, regular calling convention, presence of signature field
  case (3<<29): ABI.2010.3.16, stret calling convention, presence of signature field,
}
```

The signature field is not always populated.

The following discussions are presented as 10.6.ABI otherwise.

`Block` literals may occur within functions where the structure is created in stack local memory. They may also appear as initialization expressions for `Block` variables of global or `static` local variables.

When a `Block` literal expression is evaluated the stack based structure is initialized as follows:

1. A `static` descriptor structure is declared and initialized as follows:

   a. The `invoke` function pointer is set to a function that takes the `Block` structure as its first argument and the rest of the arguments (if any) to the `Block` and executes the `Block` compound statement.

   b. The `size` field is set to the size of the following `Block` literal structure.

c. The `copy_helper` and `dispose_helper` function pointers are set to respective helper functions if they are required by the `Block` literal.

2. A stack (or global) `Block` literal data structure is created and initialized as follows:

   a. The `isa` field is set to the address of the external `_NSConcreteStackBlock`, which is a block of uninitialized memory supplied in `libSystem`, or `_NSConcreteGlobalBlock` if this is a static or file level `Block` literal.

   b. The `flags` field is set to zero unless there are variables imported into the `Block` that need helper functions for program level `Block_copy()` and `Block_release()` operations, in which case the (1<<25) flags bit is set.

As an example, the `Block` literal expression:

```
^ { printf("hello world\n"); }
```

would cause the following to be created on a 32-bit system:

```
struct __block_literal_1 {
    void *isa;
    int flags;
    int reserved;
    void (*invoke)(struct __block_literal_1 *);
    struct __block_descriptor_1 *descriptor;
};

void __block_invoke_1(struct __block_literal_1 *_block) {
    printf("hello world\n");
}

static struct __block_descriptor_1 {
    unsigned long int reserved;
    unsigned long int Block_size;
} __block_descriptor_1 = { 0, sizeof(struct __block_literal_1) };
```

and where the `Block` literal itself appears:

```
struct __block_literal_1 _block_literal = {
     &_NSConcreteStackBlock,
     (1<<29), <uninitialized>,
     __block_invoke_1,
     &__block_descriptor_1
};
```

A `Block` imports other `Block` references, `const` copies of other variables, and variables marked `__block`. In Objective-C, variables may additionally be objects.

When a `Block` literal expression is used as the initial value of a global or `static` local variable, it is initialized as follows:

```
struct __block_literal_1 __block_literal_1 = {
      &_NSConcreteGlobalBlock,
      (1<<28)|(1<<29), <uninitialized>,
      __block_invoke_1,
      &__block_descriptor_1
};
```

that is, a different address is provided as the first value and a particular (1<<28) bit is set in the `flags` field, and otherwise it is the same as for stack based `Block` literals. This is an optimization that can be used for any `Block` literal that imports no `const` or `__block` storage variables.

Imported Variables

Variables of `auto` storage class are imported as `const` copies. Variables of `__block` storage class are imported as a pointer to an enclosing data structure. Global variables are simply referenced and not considered as imported.

Imported `const` copy variables

Automatic storage variables not marked with `__block` are imported as `const` copies.

The simplest example is that of importing a variable of type `int`:

```
int x = 10;
void (^vv)(void) = ^{ printf("x is %d\n", x); }
x = 11;
vv();
```

which would be compiled to:

```
struct __block_literal_2 {
    void *isa;
    int flags;
    int reserved;
    void (*invoke)(struct __block_literal_2 *);
    struct __block_descriptor_2 *descriptor;
    const int x;
};

void __block_invoke_2(struct __block_literal_2 *_block) {
    printf("x is %d\n", _block->x);
}

static struct __block_descriptor_2 {
    unsigned long int reserved;
    unsigned long int Block_size;
} __block_descriptor_2 = { 0, sizeof(struct __block_literal_2) };
```

and:

```
struct __block_literal_2 __block_literal_2 = {
      &_NSConcreteStackBlock,
      (1<<29), <uninitialized>,
      __block_invoke_2,
      &__block_descriptor_2,
      x
 };
```

In summary, scalars, structures, unions, and function pointers are generally imported as `const` copies with no need for helper functions.

Imported `const` copy of `Block` reference

The first case where copy and dispose helper functions are required is for the case of when a `Block` itself is imported. In this case both a `copy_helper` function and a `dispose_helper` function are needed. The `copy_helper` function is passed both the existing stack based pointer and the pointer to the new heap version and should call back into the runtime to actually do the copy operation on the imported fields within the `Block`. The runtime functions are all described in Runtime Helper Functions.

A quick example:

```
void (^existingBlock)(void) = ...;
void (^vv)(void) = ^{ existingBlock(); }
vv();

struct __block_literal_3 {
   ...; // existing block
};

struct __block_literal_4 {
    void *isa;
    int flags;
```

```
    int reserved;
    void (*invoke)(struct __block_literal_4 *);
    struct __block_literal_3 *const existingBlock;
};

void __block_invoke_4(struct __block_literal_2 *_block) {
   __block->existingBlock->invoke(__block->existingBlock);
}

void __block_copy_4(struct __block_literal_4 *dst, struct __block_literal_4 *src) {
     //_Block_copy_assign(&dst->existingBlock, src->existingBlock, 0);
     _Block_object_assign(&dst->existingBlock, src->existingBlock, BLOCK_FIELD_IS_BLOCK);
}

void __block_dispose_4(struct __block_literal_4 *src) {
     // was _Block_destroy
     _Block_object_dispose(src->existingBlock, BLOCK_FIELD_IS_BLOCK);
}

static struct __block_descriptor_4 {
    unsigned long int reserved;
    unsigned long int Block_size;
    void (*copy_helper)(struct __block_literal_4 *dst, struct __block_literal_4 *src);
    void (*dispose_helper)(struct __block_literal_4 *);
} __block_descriptor_4 = {
    0,
    sizeof(struct __block_literal_4),
    __block_copy_4,
    __block_dispose_4,
};
```

and where said `Block` is used:

```
struct __block_literal_4 _block_literal = {
      &_NSConcreteStackBlock,
      (1<<25)|(1<<29), <uninitialized>
      __block_invoke_4,
      & __block_descriptor_4
      existingBlock,
};
```

Importing `__attribute__((NSObject))` variables

GCC introduces `__attribute__((NSObject))` on structure pointers to mean “this is an object”. This is useful because many low level data structures are declared as opaque structure pointers, e.g. `CFStringRef`, `CFArrayRef`, etc. When used from C, however, these are still really objects and are the second case where that requires copy and dispose helper functions to be generated. The copy helper functions generated by the compiler should use the `_Block_object_assign` runtime helper function and in the dispose helper the `_Block_object_dispose` runtime helper function should be called.

For example, `Block` foo in the following:

```
struct Opaque *__attribute__((NSObject)) objectPointer = ...;
...
void (^foo)(void) = ^{  CFPrint(objectPointer); };
```

would have the following helper functions generated:

```
void __block_copy_foo(struct __block_literal_5 *dst, struct __block_literal_5 *src) {
     _Block_object_assign(&dst->objectPointer, src-> objectPointer, BLOCK_FIELD_IS_OBJECT);
}

void __block_dispose_foo(struct __block_literal_5 *src) {
```

```
     _Block_object_dispose(src->objectPointer, BLOCK_FIELD_IS_OBJECT);
}
```

Imported __block marked variables

Layout of __block marked variables

The compiler must embed variables that are marked `__block` in a specialized structure of the form:

```
struct _block_byref_foo {
    void *isa;
    struct Block_byref *forwarding;
    int flags;   //refcount;
    int size;
    typeof(marked_variable) marked_variable;
};
```

Variables of certain types require helper functions for when `Block_copy()` and `Block_release()` are performed upon a referencing `Block`. At the "C" level only variables that are of type `Block` or ones that have `__attribute__((NSObject))` marked require helper functions. In Objective-C objects require helper functions and in C++ stack based objects require helper functions. Variables that require helper functions use the form:

```
struct _block_byref_foo {
    void *isa;
    struct _block_byref_foo *forwarding;
    int flags;   //refcount;
    int size;
    // helper functions called via Block_copy() and Block_release()
    void (*byref_keep)(void  *dst, void *src);
    void (*byref_dispose)(void *);
    typeof(marked_variable) marked_variable;
};
```

The structure is initialized such that:

a. The `forwarding` pointer is set to the beginning of its enclosing structure.

b. The `size` field is initialized to the total size of the enclosing structure.

c. The `flags` field is set to either 0 if no helper functions are needed or (1<<25) if they are.

d. The helper functions are initialized (if present).

e. The variable itself is set to its initial value.

f. The `isa` field is set to `NULL`.

Access to __block variables from within its lexical scope

In order to "move" the variable to the heap upon a `copy_helper` operation the compiler must rewrite access to such a variable to be indirect through the structures `forwarding` pointer. For example:

```
int __block i = 10;
i = 11;
```

would be rewritten to be:

```
struct _block_byref_i {
  void *isa;
  struct _block_byref_i *forwarding;
  int flags;   //refcount;
  int size;
  int captured_i;
} i = { NULL, &i, 0, sizeof(struct _block_byref_i), 10 };

i.forwarding->captured_i = 11;
```

In the case of a `Block` reference variable being marked `__block` the helper code generated must use the `_Block_object_assign` and `_Block_object_dispose` routines supplied by the runtime to make the copies. For example:

```
__block void (voidBlock)(void) = blockA;
voidBlock = blockB;
```

would translate into:

```
struct _block_byref_voidBlock {
    void *isa;
    struct _block_byref_voidBlock *forwarding;
    int flags;   //refcount;
    int size;
    void (*byref_keep)(struct _block_byref_voidBlock *dst, struct _block_byref_voidBlock *src);
    void (*byref_dispose)(struct _block_byref_voidBlock *);
    void (^captured_voidBlock)(void);
};

void _block_byref_keep_helper(struct _block_byref_voidBlock *dst, struct _block_byref_voidBlock *src) {
    //_Block_copy_assign(&dst->captured_voidBlock, src->captured_voidBlock, 0);
    _Block_object_assign(&dst->captured_voidBlock, src->captured_voidBlock, BLOCK_FIELD_IS_BLOCK | BLOCK_BYREF_CALLER);
}

void _block_byref_dispose_helper(struct _block_byref_voidBlock *param) {
    //_Block_destroy(param->captured_voidBlock, 0);
    _Block_object_dispose(param->captured_voidBlock, BLOCK_FIELD_IS_BLOCK | BLOCK_BYREF_CALLER)}
```

and:

```
struct _block_byref_voidBlock voidBlock = {( .forwarding=&voidBlock, .flags=(1<<25), .size=sizeof(struct _block_byref_voidBlock *),
    .byref_keep=_block_byref_keep_helper, .byref_dispose=_block_byref_dispose_helper,
    .captured_voidBlock=blockA )};

voidBlock.forwarding->captured_voidBlock = blockB;
```

Importing `__block` variables into `Blocks`

A `Block` that uses a `__block` variable in its compound statement body must import the variable and emit `copy_helper` and `dispose_helper` helper functions that, in turn, call back into the runtime to actually copy or release the `byref` data block using the functions `_Block_object_assign` and `_Block_object_dispose`.

For example:

```
int __block i = 2;
functioncall(^{ i = 10; });
```

would translate to:

```
struct _block_byref_i {
    void *isa;  // set to NULL
    struct _block_byref_voidBlock *forwarding;
    int flags;   //refcount;
    int size;
    void (*byref_keep)(struct _block_byref_i *dst, struct _block_byref_i *src);
    void (*byref_dispose)(struct _block_byref_i *);
    int captured_i;
};


struct __block_literal_5 {
    void *isa;
    int flags;
    int reserved;
    void (*invoke)(struct __block_literal_5 *);
    struct __block_descriptor_5 *descriptor;
    struct _block_byref_i *i_holder;
};

void __block_invoke_5(struct __block_literal_5 *_block) {
   _block->forwarding->captured_i = 10;
}
```

```
void __block_copy_5(struct __block_literal_5 *dst, struct __block_literal_5 *src) {
     //_Block_byref_assign_copy(&dst->captured_i, src->captured_i);
     _Block_object_assign(&dst->captured_i, src->captured_i, BLOCK_FIELD_IS_BYREF | BLOCK_BYREF_CALLER);
}

void __block_dispose_5(struct __block_literal_5 *src) {
     //_Block_byref_release(src->captured_i);
     _Block_object_dispose(src->captured_i, BLOCK_FIELD_IS_BYREF | BLOCK_BYREF_CALLER);
}

static struct __block_descriptor_5 {
    unsigned long int reserved;
    unsigned long int Block_size;
    void (*copy_helper)(struct __block_literal_5 *dst, struct __block_literal_5 *src);
    void (*dispose_helper)(struct __block_literal_5 *);
} __block_descriptor_5 = { 0, sizeof(struct __block_literal_5) __block_copy_5, __block_dispose_5 };
```

and:

```
struct _block_byref_i i = {( .isa=NULL, .forwarding=&i, .flags=0, .size=sizeof(struct _block_byref_i), .captured_i=2 )};
struct __block_literal_5 _block_literal = {
      &_NSConcreteStackBlock,
      (1<<25)|(1<<29), <uninitialized>,
      __block_invoke_5,
      &__block_descriptor_5,
      &i,
};
```

Importing `__attribute__((NSObject))` `__block` variables

A `__block` variable that is also marked `__attribute__((NSObject))` should have `byref_keep` and `byref_dispose` helper functions that use `_Block_object_assign` and `_Block_object_dispose`.

`__block` escapes

Because `Blocks` referencing `__block` variables may have `Block_copy()` performed upon them the underlying storage for the variables may move to the heap. In Objective-C Garbage Collection Only compilation environments the heap used is the garbage collected one and no further action is required. Otherwise the compiler must issue a call to potentially release any heap storage for `__block` variables at all escapes or terminations of their scope. The call should be:

```
_Block_object_dispose(&_block_byref_foo, BLOCK_FIELD_IS_BYREF);
```

Nesting

`Blocks` may contain `Block` literal expressions. Any variables used within inner blocks are imported into all enclosing `Block` scopes even if the variables are not used. This includes `const` imports as well as `__block` variables.

Objective C Extensions to `Blocks`

Importing Objects

Objects should be treated as `__attribute__((NSObject))` variables; all `copy_helper`, `dispose_helper`, `byref_keep`, and `byref_dispose` helper functions should use `_Block_object_assign` and `_Block_object_dispose`. There should be no code generated that uses `*-retain` or `*-release` methods.

`Blocks` as Objects

The compiler will treat `Blocks` as objects when synthesizing property setters and getters, will characterize them as objects when generating garbage collection strong and weak layout information in the same manner as objects, and will issue strong and weak write-barrier assignments in the same manner as objects.

`__weak __block` Support

Objective-C (and Objective-C++) support the `__weak` attribute on `__block` variables. Under normal circumstances the compiler uses the Objective-C runtime helper support functions `objc_assign_weak` and `objc_read_weak`. Both should continue to be used for all reads and writes of `__weak __block` variables:

```
objc_read_weak(&block->byref_i->forwarding->i)
```

The `__weak` variable is stored in a `_block_byref_foo` structure and the `Block` has copy and dispose helpers for this structure that call:

```
_Block_object_assign(&dest->_block_byref_i, src-> _block_byref_i, BLOCK_FIELD_IS_WEAK | BLOCK_FIELD_IS_BYREF);
```

and:

```
_Block_object_dispose(src->_block_byref_i, BLOCK_FIELD_IS_WEAK | BLOCK_FIELD_IS_BYREF);
```

In turn, the `block_byref` copy support helpers distinguish between whether the `__block` variable is a `Block` or not and should either call:

```
_Block_object_assign(&dest->_block_byref_i, src->_block_byref_i, BLOCK_FIELD_IS_WEAK | BLOCK_FIELD_IS_OBJECT | BLOCK_BYREF_CALLER);
```

for something declared as an object or:

```
_Block_object_assign(&dest->_block_byref_i, src->_block_byref_i, BLOCK_FIELD_IS_WEAK | BLOCK_FIELD_IS_BLOCK | BLOCK_BYREF_CALLER);
```

for something declared as a `Block`.

A full example follows:

```
__block __weak id obj = <initialization expression>;
functioncall(^{ [obj somemessage]; });
```

would translate to:

```
struct _block_byref_obj {
    void *isa;  // uninitialized
    struct _block_byref_obj *forwarding;
    int flags;   //refcount;
    int size;
    void (*byref_keep)(struct _block_byref_i *dst, struct _block_byref_i *src);
    void (*byref_dispose)(struct _block_byref_i *);
    id captured_obj;
};

void _block_byref_obj_keep(struct _block_byref_voidBlock *dst, struct _block_byref_voidBlock *src) {
    //_Block_copy_assign(&dst->captured_obj, src->captured_obj, 0);
    _Block_object_assign(&dst->captured_obj, src->captured_obj, BLOCK_FIELD_IS_OBJECT | BLOCK_FIELD_IS_WEAK | BLOCK_BYREF_CALLER);
}

void _block_byref_obj_dispose(struct _block_byref_voidBlock *param) {
    //_Block_destroy(param->captured_obj, 0);
    _Block_object_dispose(param->captured_obj, BLOCK_FIELD_IS_OBJECT | BLOCK_FIELD_IS_WEAK | BLOCK_BYREF_CALLER);
};
```

for the block `byref` part and:

```
struct __block_literal_5 {
    void *isa;
    int flags;
    int reserved;
    void (*invoke)(struct __block_literal_5 *);
    struct __block_descriptor_5 *descriptor;
    struct _block_byref_obj *byref_obj;
};

void __block_invoke_5(struct __block_literal_5 *_block) {
   [objc_read_weak(&_block->byref_obj->forwarding->captured_obj) somemessage];
}

void __block_copy_5(struct __block_literal_5 *dst, struct __block_literal_5 *src) {
     //_Block_byref_assign_copy(&dst->byref_obj, src->byref_obj);
     _Block_object_assign(&dst->byref_obj, src->byref_obj, BLOCK_FIELD_IS_BYREF | BLOCK_FIELD_IS_WEAK);
}


void __block_dispose_5(struct __block_literal_5 *src) {
     //_Block_byref_release(src->byref_obj);
```

```
     _Block_object_dispose(src->byref_obj, BLOCK_FIELD_IS_BYREF | BLOCK_FIELD_IS_WEAK);
}

static struct __block_descriptor_5 {
    unsigned long int reserved;
    unsigned long int Block_size;
    void (*copy_helper)(struct __block_literal_5 *dst, struct __block_literal_5 *src);
    void (*dispose_helper)(struct __block_literal_5 *);
} __block_descriptor_5 = { 0, sizeof(struct __block_literal_5), __block_copy_5, __block_dispose_5 };
```

and within the compound statement:

```
truct _block_byref_obj obj = {( .forwarding=&obj, .flags=(1<<25), .size=sizeof(struct _block_byref_obj),
                .byref_keep=_block_byref_obj_keep, .byref_dispose=_block_byref_obj_dispose,
                .captured_obj = <initialization expression> )};

truct __block_literal_5 _block_literal = {
     &_NSConcreteStackBlock,
     (1<<25)|(1<<29), <uninitialized>,
     __block_invoke_5,
     &__block_descriptor_5,
     &obj,        // a reference to the on-stack structure containing "captured_obj"
};


functioncall(_block_literal->invoke(&_block_literal));
```

C++ Support

Within a block stack based C++ objects are copied into `const` copies using the copy constructor. It is an error if a stack based C++ object is used within a block if it does not have a copy constructor. In addition both copy and destroy helper routines must be synthesized for the block to support the `Block_copy()` operation, and the flags work marked with the (1<<26) bit in addition to the (1<<25) bit. The copy helper should call the constructor using appropriate offsets of the variable within the supplied stack based block source and heap based destination for all `const` constructed copies, and similarly should call the destructor in the destroy routine.

As an example, suppose a C++ class `FOO` existed with a copy constructor. Within a code block a stack version of a `FOO` object is declared and used within a `Block` literal expression:

```
{
     FOO foo;
     void (^block)(void) = ^{ printf("%d\n", foo.value()); };
}
```

The compiler would synthesize:

```
struct __block_literal_10 {
    void *isa;
    int flags;
    int reserved;
    void (*invoke)(struct __block_literal_10 *);
    struct __block_descriptor_10 *descriptor;
    const FOO foo;
};

void __block_invoke_10(struct __block_literal_10 *_block) {
   printf("%d\n", _block->foo.value());
}

void __block_copy_10(struct __block_literal_10 *dst, struct __block_literal_10 *src) {
     FOO_ctor(&dst->foo, &src->foo);
}

void __block_dispose_10(struct __block_literal_10 *src) {
     FOO_dtor(&src->foo);
}


static struct __block_descriptor_10 {
     unsigned long int reserved;
```

```
    unsigned long int Block_size;
    void (*copy_helper)(struct __block_literal_10 *dst, struct __block_literal_10 *src);
    void (*dispose_helper)(struct __block_literal_10 *);
} __block_descriptor_10 = { 0, sizeof(struct __block_literal_10), __block_copy_10, __block_dispose_10 };
```

and the code would be:

```
{
  FOO foo;
  comp_ctor(&foo); // default constructor
  struct __block_literal_10 _block_literal = {
    &_NSConcreteStackBlock,
    (1<<25)|(1<<26)|(1<<29), <uninitialized>,
    __block_invoke_10,
    &__block_descriptor_10,
   };
   comp_ctor(&_block_literal->foo, &foo);  // const copy into stack version
   struct __block_literal_10 &block = &_block_literal;  // assign literal to block variable
   block->invoke(block);    // invoke block
   comp_dtor(&_block_literal->foo); // destroy stack version of const block copy
   comp_dtor(&foo); // destroy original version
}
```

C++ objects stored in `__block` storage start out on the stack in a `block_byref` data structure as do other variables. Such objects (if not `const` objects) must support a regular copy constructor. The `block_byref` data structure will have copy and destroy helper routines synthesized by the compiler. The copy helper will have code created to perform the copy constructor based on the initial stack `block_byref` data structure, and will also set the (1<<26) bit in addition to the (1<<25) bit. The destroy helper will have code to do the destructor on the object stored within the supplied `block_byref` heap data structure. For example,

```
__block FOO blockStorageFoo;
```

requires the normal constructor for the embedded `blockStorageFoo` object:

```
FOO_ctor(& _block_byref_blockStorageFoo->blockStorageFoo);
```

and at scope termination the destructor:

```
FOO_dtor(& _block_byref_blockStorageFoo->blockStorageFoo);
```

Note that the forwarding indirection is *NOT* used.

The compiler would need to generate (if used from a block literal) the following copy/dispose helpers:

```
void _block_byref_obj_keep(struct _block_byref_blockStorageFoo *dst, struct _block_byref_blockStorageFoo *src) {
     FOO_ctor(&dst->blockStorageFoo, &src->blockStorageFoo);
}

void _block_byref_obj_dispose(struct _block_byref_blockStorageFoo *src) {
     FOO_dtor(&src->blockStorageFoo);
}
```

for the appropriately named constructor and destructor for the class/struct `FOO`.

To support member variable and function access the compiler will synthesize a `const` pointer to a block version of the `this` pointer.

Runtime Helper Functions

The runtime helper functions are described in `/usr/local/include/Block_private.h`. To summarize their use, a `Block` requires copy/dispose helpers if it imports any block variables, `__block` storage variables, `__attribute__((NSObject))` variables, or C++ `const` copied objects with constructor/destructors. The (1<<26) bit is set and functions are generated.

The block copy helper function should, for each of the variables of the type mentioned above, call:

```
_Block_object_assign(&dst->target, src->target, BLOCK_FIELD_<apropos>);
```

in the copy helper and:

```
_Block_object_dispose(->target, BLOCK_FIELD_<apropos>);
```

in the dispose helper where <apropos> is:

```
enum {
    BLOCK_FIELD_IS_OBJECT   =  3,  // id, NSObject, __attribute__((NSObject)), block, ...
    BLOCK_FIELD_IS_BLOCK    =  7,  // a block variable
    BLOCK_FIELD_IS_BYREF    =  8,  // the on stack structure holding the __block variable

    BLOCK_FIELD_IS_WEAK     = 16,  // declared __weak

    BLOCK_BYREF_CALLER      = 128, // called from byref copy/dispose helpers
};
```

and of course the constructors/destructors for `const` copied C++ objects.

The `block_byref` data structure similarly requires copy/dispose helpers for block variables, `__attribute__((NSObject))` variables, or C++ `const` copied objects with constructor/destructors, and again the (1<<26) bit is set and functions are generated in the same manner.

Under ObjC we allow `__weak` as an attribute on `__block` variables, and this causes the addition of `BLOCK_FIELD_IS_WEAK` orred onto the `BLOCK_FIELD_IS_BYREF` flag when copying the `block_byref` structure in the `Block` copy helper, and onto the `BLOCK_FIELD_<apropos>` field within the `block_byref` copy/dispose helper calls.

The prototypes, and summary, of the helper functions are:

```
/* Certain field types require runtime assistance when being copied to the
   heap.  The following function is used to copy fields of types: blocks,
   pointers to byref structures, and objects (including
   __attribute__((NSObject)) pointers.  BLOCK_FIELD_IS_WEAK is orthogonal to
   the other choices which are mutually exclusive.  Only in a Block copy
   helper will one see BLOCK_FIELD_IS_BYREF.
*/
void _Block_object_assign(void *destAddr, const void *object, const int flags);

/* Similarly a compiler generated dispose helper needs to call back for each
   field of the byref data structure.  (Currently the implementation only
   packs one field into the byref structure but in principle there could be
   more).  The same flags used in the copy helper should be used for each
   call generated to this function:
*/
void _Block_object_dispose(const void *object, const int flags);
```

Copyright

Objective-C Automatic Reference Counting (ARC)

About this document 89
- Purpose 89
- Background 89
- Evolution 90

General 91

Retainable object pointers 91
- Retain count semantics 92
- Retainable object pointers as operands and arguments 92
  - Consumed parameters 93
  - Retained return values 93
  - Unretained return values 94
  - Bridged casts 95
- Restrictions 95
  - Conversion of retainable object pointers 95
  - Conversion to retainable object pointer type of expressions with known semantics 95
  - Conversion from retainable object pointer type in certain contexts 96

Ownership qualification 96
- Spelling 97
  - Property declarations 98
- Semantics 98
- Restrictions 99
  - Weak-unavailable types 99
  - Storage duration of `__autoreleasing` objects 99
  - Conversion of pointers to ownership-qualified types 100
  - Passing to an out parameter by writeback 101
  - Ownership-qualified fields of structs and unions 102
    - Formal rules for non-trivial types in C 102
    - Application of the formal C rules to nontrivial ownership qualifiers 104
    - C/C++ compatibility for structs and unions with non-trivial members 104
- Ownership inference 105
  - Objects 105
  - Indirect parameters 105
  - Template arguments 106

Method families 106
- Explicit method family control 107
- Semantics of method families 107
  - Semantics of `init` 107
  - Related result types 108

Optimization 108
- Object liveness 109
- No object lifetime extension 110
- Precise lifetime semantics 110

Miscellaneous 111

Special methods 111
Memory management methods 111
dealloc 111
@autoreleasepool 112
Externally-Retained Variables 113
self 113
Fast enumeration iteration variables 114
Blocks 114
Exceptions 114
Interior pointers 115
C retainable pointer types 115
Auditing of C retainable pointer interfaces 116
Runtime support 116
id objc_autorelease(id value); 117
void objc_autoreleasePoolPop(void *pool); 117
void *objc_autoreleasePoolPush(void); 117
id objc_autoreleaseReturnValue(id value); 117
void objc_copyWeak(id *dest, id *src); 118
void objc_destroyWeak(id *object); 118
id objc_initWeak(id *object, id value); 118
id objc_loadWeak(id *object); 118
id objc_loadWeakRetained(id *object); 119
void objc_moveWeak(id *dest, id *src); 119
void objc_release(id value); 119
id objc_retain(id value); 119
id objc_retainAutorelease(id value); 119
id objc_retainAutoreleaseReturnValue(id value); 119
id objc_retainAutoreleasedReturnValue(id value); 119
id objc_retainBlock(id value); 120
void objc_storeStrong(id *object, id value); 120
id objc_storeWeak(id *object, id value); 120
id objc_unsafeClaimAutoreleasedReturnValue(id value); 120

About this document

Purpose

The first and primary purpose of this document is to serve as a complete technical specification of Automatic Reference Counting. Given a core Objective-C compiler and runtime, it should be possible to write a compiler and runtime which implements these new semantics.

The secondary purpose is to act as a rationale for why ARC was designed in this way. This should remain tightly focused on the technical design and should not stray into marketing speculation.

Background

This document assumes a basic familiarity with C.

Blocks are a C language extension for creating anonymous functions. Users interact with and transfer block objects using block pointers, which are represented like a normal pointer. A block may capture values from local variables;

when this occurs, memory must be dynamically allocated. The initial allocation is done on the stack, but the runtime provides a `Block_copy` function which, given a block pointer, either copies the underlying block object to the heap, setting its reference count to 1 and returning the new block pointer, or (if the block object is already on the heap) increases its reference count by 1. The paired function is `Block_release`, which decreases the reference count by 1 and destroys the object if the count reaches zero and is on the heap.

Objective-C is a set of language extensions, significant enough to be considered a different language. It is a strict superset of C. The extensions can also be imposed on C++, producing a language called Objective-C++. The primary feature is a single-inheritance object system; we briefly describe the modern dialect.

Objective-C defines a new type kind, collectively called the object pointer types. This kind has two notable builtin members, `id` and `Class`; `id` is the final supertype of all object pointers. The validity of conversions between object pointer types is not checked at runtime. Users may define classes; each class is a type, and the pointer to that type is an object pointer type. A class may have a superclass; its pointer type is a subtype of its superclass's pointer type. A class has a set of ivars, fields which appear on all instances of that class. For every class *T* there's an associated metaclass; it has no fields, its superclass is the metaclass of *T*'s superclass, and its metaclass is a global class. Every class has a global object whose class is the class's metaclass; metaclasses have no associated type, so pointers to this object have type `Class`.

A class declaration (`@interface`) declares a set of methods. A method has a return type, a list of argument types, and a selector: a name like `foo:bar:baz:`, where the number of colons corresponds to the number of formal arguments. A method may be an instance method, in which case it can be invoked on objects of the class, or a class method, in which case it can be invoked on objects of the metaclass. A method may be invoked by providing an object (called the receiver) and a list of formal arguments interspersed with the selector, like so:

```
[receiver foo: fooArg bar: barArg baz: bazArg]
```

This looks in the dynamic class of the receiver for a method with this name, then in that class's superclass, etc., until it finds something it can execute. The receiver "expression" may also be the name of a class, in which case the actual receiver is the class object for that class, or (within method definitions) it may be `super`, in which case the lookup algorithm starts with the static superclass instead of the dynamic class. The actual methods dynamically found in a class are not those declared in the `@interface`, but those defined in a separate `@implementation` declaration; however, when compiling a call, typechecking is done based on the methods declared in the `@interface`.

Method declarations may also be grouped into protocols, which are not inherently associated with any class, but which classes may claim to follow. Object pointer types may be qualified with additional protocols that the object is known to support.

Class extensions are collections of ivars and methods, designed to allow a class's `@interface` to be split across multiple files; however, there is still a primary implementation file which must see the `@interfaces` of all class extensions. Categories allow methods (but not ivars) to be declared *post hoc* on an arbitrary class; the methods in the category's `@implementation` will be dynamically added to that class's method tables which the category is loaded at runtime, replacing those methods in case of a collision.

In the standard environment, objects are allocated on the heap, and their lifetime is manually managed using a reference count. This is done using two instance methods which all classes are expected to implement: `retain` increases the object's reference count by `1`, whereas `release` decreases it by `1` and calls the instance method `dealloc` if the count reaches 0. To simplify certain operations, there is also an autorelease pool, a thread-local list of objects to call `release` on later; an object can be added to this pool by calling `autorelease` on it.

Block pointers may be converted to type `id`; block objects are laid out in a way that makes them compatible with Objective-C objects. There is a builtin class that all block objects are considered to be objects of; this class implements `retain` by adjusting the reference count, not by calling `Block_copy`.

Evolution

ARC is under continual evolution, and this document must be updated as the language progresses.

If a change increases the expressiveness of the language, for example by lifting a restriction or by adding new syntax, the change will be annotated with a revision marker, like so:

> ARC applies to Objective-C pointer types, block pointer types, and [beginning Apple 8.0, LLVM 3.8] BPTRs declared within `extern "BCPL"` blocks.

For now, it is sensible to version this document by the releases of its sole implementation (and its host project), clang. “LLVM X.Y” refers to an open-source release of clang from the LLVM project. “Apple X.Y” refers to an Apple-provided release of the Apple LLVM Compiler. Other organizations that prepare their own, separately-versioned clang releases and wish to maintain similar information in this document should send requests to cfe-dev.

If a change decreases the expressiveness of the language, for example by imposing a new restriction, this should be taken as an oversight in the original specification and something to be avoided in all versions. Such changes are generally to be avoided.

General

Automatic Reference Counting implements automatic memory management for Objective-C objects and blocks, freeing the programmer from the need to explicitly insert retains and releases. It does not provide a cycle collector; users must explicitly manage the lifetime of their objects, breaking cycles manually or with weak or unsafe references.

ARC may be explicitly enabled with the compiler flag `-fobjc-arc`. It may also be explicitly disabled with the compiler flag `-fno-objc-arc`. The last of these two flags appearing on the compile line “wins”.

If ARC is enabled, `__has_feature(objc_arc)` will expand to 1 in the preprocessor. For more information about `__has_feature`, see the language extensions document.

Retainable object pointers

This section describes retainable object pointers, their basic operations, and the restrictions imposed on their use under ARC. Note in particular that it covers the rules for pointer *values* (patterns of bits indicating the location of a pointed-to object), not pointer *objects* (locations in memory which store pointer values). The rules for objects are covered in the next section.

A retainable object pointer (or “retainable pointer”) is a value of a retainable object pointer type (“retainable type”). There are three kinds of retainable object pointer types:

- block pointers (formed by applying the caret (^) declarator sigil to a function type)
- Objective-C object pointers (`id`, `Class`, `NSFoo*`, etc.)
- typedefs marked with `__attribute__((NSObject))`

Other pointer types, such as `int*` and `CFStringRef`, are not subject to ARC’s semantics and restrictions.

Rationale

We are not at liberty to require all code to be recompiled with ARC; therefore, ARC must interoperate with Objective-C code which manages retains and releases manually. In general, there are three requirements in order for a compiler-supported reference-count system to provide reliable interoperation:

- The type system must reliably identify which objects are to be managed. An `int*` might be a pointer to a `malloc`’ed array, or it might be an interior pointer to such an array, or it might point to some field or local variable. In contrast, values of the retainable object pointer types are never interior.
- The type system must reliably indicate how to manage objects of a type. This usually means that the type must imply a procedure for incrementing and decrementing retain counts. Supporting single-ownership objects requires a lot more explicit mediation in the language.
- There must be reliable conventions for whether and when “ownership” is passed between caller and callee, for both arguments and return values. Objective-C methods follow such a convention very reliably, at least for system libraries on macOS, and functions always pass objects at +0. The C-based APIs for Core Foundation objects, on the other hand, have much more varied transfer semantics.

The use of `__attribute__((NSObject))` typedefs is not recommended. If it’s absolutely necessary to use this attribute, be very explicit about using the typedef, and do not assume that it will be preserved by language features like `__typeof` and C++ template argument substitution.

Rationale

Any compiler operation which incidentally strips type “sugar” from a type will yield a type without the attribute, which may result in unexpected behavior.

Retain count semantics

A retainable object pointer is either a null pointer or a pointer to a valid object. Furthermore, if it has block pointer type and is not `null` then it must actually be a pointer to a block object, and if it has `Class` type (possibly protocol-qualified) then it must actually be a pointer to a class object. Otherwise ARC does not enforce the Objective-C type system as long as the implementing methods follow the signature of the static type. It is undefined behavior if ARC is exposed to an invalid pointer.

For ARC’s purposes, a valid object is one with “well-behaved” retaining operations. Specifically, the object must be laid out such that the Objective-C message send machinery can successfully send it the following messages:

- `retain`, taking no arguments and returning a pointer to the object.
- `release`, taking no arguments and returning `void`.
- `autorelease`, taking no arguments and returning a pointer to the object.

The behavior of these methods is constrained in the following ways. The term high-level semantics is an intentionally vague term; the intent is that programmers must implement these methods in a way such that the compiler, modifying code in ways it deems safe according to these constraints, will not violate their requirements. For example, if the user puts logging statements in `retain`, they should not be surprised if those statements are executed more or less often depending on optimization settings. These constraints are not exhaustive of the optimization opportunities: values held in local variables are subject to additional restrictions, described later in this document.

It is undefined behavior if a computation history featuring a send of `retain` followed by a send of `release` to the same object, with no intervening `release` on that object, is not equivalent under the high-level semantics to a computation history in which these sends are removed. Note that this implies that these methods may not raise exceptions.

It is undefined behavior if a computation history features any use whatsoever of an object following the completion of a send of `release` that is not preceded by a send of `retain` to the same object.

The behavior of `autorelease` must be equivalent to sending `release` when one of the autorelease pools currently in scope is popped. It may not throw an exception.

When the semantics call for performing one of these operations on a retainable object pointer, if that pointer is `null` then the effect is a no-op.

All of the semantics described in this document are subject to additional optimization rules which permit the removal or optimization of operations based on local knowledge of data flow. The semantics describe the high-level behaviors that the compiler implements, not an exact sequence of operations that a program will be compiled into.

Retainable object pointers as operands and arguments

In general, ARC does not perform retain or release operations when simply using a retainable object pointer as an operand within an expression. This includes:

- loading a retainable pointer from an object with non-weak ownership,
- passing a retainable pointer as an argument to a function or method, and
- receiving a retainable pointer as the result of a function or method call.

Rationale

While this might seem uncontroversial, it is actually unsafe when multiple expressions are evaluated in “parallel”, as with binary operators and calls, because (for example) one expression might load from an object while another writes to it. However, C and C++ already call this undefined behavior because the evaluations are unsequenced, and ARC simply exploits that here to avoid needing to retain arguments across a large number of calls.

The remainder of this section describes exceptions to these rules, how those exceptions are detected, and what those exceptions imply semantically.

Consumed parameters

A function or method parameter of retainable object pointer type may be marked as consumed, signifying that the callee expects to take ownership of a +1 retain count. This is done by adding the `ns_consumed` attribute to the parameter declaration, like so:

```
void foo(__attribute((ns_consumed)) id x);
- (void) foo: (id) __attribute((ns_consumed)) x;
```

This attribute is part of the type of the function or method, not the type of the parameter. It controls only how the argument is passed and received.

When passing such an argument, ARC retains the argument prior to making the call.

When receiving such an argument, ARC releases the argument at the end of the function, subject to the usual optimizations for local values.

Rationale

This formalizes direct transfers of ownership from a caller to a callee. The most common scenario here is passing the `self` parameter to `init`, but it is useful to generalize. Typically, local optimization will remove any extra retains and releases: on the caller side the retain will be merged with a +1 source, and on the callee side the release will be rolled into the initialization of the parameter.

The implicit `self` parameter of a method may be marked as consumed by adding `__attribute__((ns_consumes_self))` to the method declaration. Methods in the `init` family are treated as if they were implicitly marked with this attribute.

It is undefined behavior if an Objective-C message send to a method with `ns_consumed` parameters (other than self) is made with a null receiver. It is undefined behavior if the method to which an Objective-C message send statically resolves to has a different set of `ns_consumed` parameters than the method it dynamically resolves to. It is undefined behavior if a block or function call is made through a static type with a different set of `ns_consumed` parameters than the implementation of the called block or function.

Rationale

Consumed parameters with null receiver are a guaranteed leak. Mismatches with consumed parameters will cause over-retains or over-releases, depending on the direction. The rule about function calls is really just an application of the existing C/C++ rule about calling functions through an incompatible function type, but it’s useful to state it explicitly.

Retained return values

A function or method which returns a retainable object pointer type may be marked as returning a retained value, signifying that the caller expects to take ownership of a +1 retain count. This is done by adding the `ns_returns_retained` attribute to the function or method declaration, like so:

```
id foo(void) __attribute((ns_returns_retained));
- (id) foo __attribute((ns_returns_retained));
```

This attribute is part of the type of the function or method.

When returning from such a function or method, ARC retains the value at the point of evaluation of the return statement, before leaving all local scopes.

When receiving a return result from such a function or method, ARC releases the value at the end of the full-expression it is contained within, subject to the usual optimizations for local values.

Rationale

This formalizes direct transfers of ownership from a callee to a caller. The most common scenario this models is the retained return from `init`, `alloc`, `new`, and `copy` methods, but there are other cases in the frameworks. After optimization there are typically no extra retains and releases required.

Methods in the `alloc`, `copy`, `init`, `mutableCopy`, and `new` families are implicitly marked `__attribute__((ns_returns_retained))`. This may be suppressed by explicitly marking the method `__attribute__((ns_returns_not_retained))`.

It is undefined behavior if the method to which an Objective-C message send statically resolves has different retain semantics on its result from the method it dynamically resolves to. It is undefined behavior if a block or function call is made through a static type with different retain semantics on its result from the implementation of the called block or function.

Rationale

Mismatches with returned results will cause over-retains or over-releases, depending on the direction. Again, the rule about function calls is really just an application of the existing C/C++ rule about calling functions through an incompatible function type.

Unretained return values

A method or function which returns a retainable object type but does not return a retained value must ensure that the object is still valid across the return boundary.

When returning from such a function or method, ARC retains the value at the point of evaluation of the return statement, then leaves all local scopes, and then balances out the retain while ensuring that the value lives across the call boundary. In the worst case, this may involve an `autorelease`, but callers must not assume that the value is actually in the autorelease pool.

ARC performs no extra mandatory work on the caller side, although it may elect to do something to shorten the lifetime of the returned value.

Rationale

It is common in non-ARC code to not return an autoreleased value; therefore the convention does not force either path. It is convenient to not be required to do unnecessary retains and autoreleases; this permits optimizations such as eliding retain/autoreleases when it can be shown that the original pointer will still be valid at the point of return.

A method or function may be marked with `__attribute__((ns_returns_autoreleased))` to indicate that it returns a pointer which is guaranteed to be valid at least as long as the innermost autorelease pool. There are no additional semantics enforced in the definition of such a method; it merely enables optimizations in callers.

Bridged casts

A bridged cast is a C-style cast annotated with one of three keywords:

- `(__bridge T) op` casts the operand to the destination type `T`. If `T` is a retainable object pointer type, then `op` must have a non-retainable pointer type. If `T` is a non-retainable pointer type, then `op` must have a retainable object pointer type. Otherwise the cast is ill-formed. There is no transfer of ownership, and ARC inserts no retain operations.
- `(__bridge_retained T) op` casts the operand, which must have retainable object pointer type, to the destination type, which must be a non-retainable pointer type. ARC retains the value, subject to the usual optimizations on local values, and the recipient is responsible for balancing that +1.
- `(__bridge_transfer T) op` casts the operand, which must have non-retainable pointer type, to the destination type, which must be a retainable object pointer type. ARC will release the value at the end of the enclosing full-expression, subject to the usual optimizations on local values.

These casts are required in order to transfer objects in and out of ARC control; see the rationale in the section on conversion of retainable object pointers.

Using a `__bridge_retained` or `__bridge_transfer` cast purely to convince ARC to emit an unbalanced retain or release, respectively, is poor form.

Restrictions

Conversion of retainable object pointers

In general, a program which attempts to implicitly or explicitly convert a value of retainable object pointer type to any non-retainable type, or vice-versa, is ill-formed. For example, an Objective-C object pointer shall not be converted to `void*`. As an exception, cast to `intptr_t` is allowed because such casts are not transferring ownership. The bridged casts may be used to perform these conversions where necessary.

Rationale

We cannot ensure the correct management of the lifetime of objects if they may be freely passed around as unmanaged types. The bridged casts are provided so that the programmer may explicitly describe whether the cast transfers control into or out of ARC.

However, the following exceptions apply.

Conversion to retainable object pointer type of expressions with known semantics

[beginning Apple 4.0, LLVM 3.1] These exceptions have been greatly expanded; they previously applied only to a much-reduced subset which is difficult to categorize but which included null pointers, message sends (under the given rules), and the various global constants.

An unbridged conversion to a retainable object pointer type from a type other than a retainable object pointer type is ill-formed, as discussed above, unless the operand of the cast has a syntactic form which is known retained, known unretained, or known retain-agnostic.

An expression is known retain-agnostic if it is:

- an Objective-C string literal,
- a load from a `const` system global variable of C retainable pointer type, or
- a null pointer constant.

An expression is known unretained if it is an rvalue of C retainable pointer type and it is:

- a direct call to a function, and either that function has the `cf_returns_not_retained` attribute or it is an audited function that does not have the `cf_returns_retained` attribute and does not follow the create/copy naming convention,
- a message send, and the declared method either has the `cf_returns_not_retained` attribute or it has neither the `cf_returns_retained` attribute nor a selector family that implies a retained result, or

- [beginning LLVM 3.6] a load from a `const` non-system global variable.

An expression is known retained if it is an rvalue of C retainable pointer type and it is:

- a message send, and the declared method either has the `cf_returns_retained` attribute, or it does not have the `cf_returns_not_retained` attribute but it does have a selector family that implies a retained result.

Furthermore:

- a comma expression is classified according to its right-hand side,
- a statement expression is classified according to its result expression, if it has one,
- an lvalue-to-rvalue conversion applied to an Objective-C property lvalue is classified according to the underlying message send, and
- a conditional operator is classified according to its second and third operands, if they agree in classification, or else the other if one is known retain-agnostic.

If the cast operand is known retained, the conversion is treated as a `__bridge_transfer` cast. If the cast operand is known unretained or known retain-agnostic, the conversion is treated as a `__bridge` cast.

Rationale

Bridging casts are annoying. Absent the ability to completely automate the management of CF objects, however, we are left with relatively poor attempts to reduce the need for a glut of explicit bridges. Hence these rules.

We've so far consciously refrained from implicitly turning retained CF results from function calls into `__bridge_transfer` casts. The worry is that some code patterns — for example, creating a CF value, assigning it to an ObjC-typed local, and then calling `CFRelease` when done — are a bit too likely to be accidentally accepted, leading to mysterious behavior.

For loads from `const` global variables of C retainable pointer type, it is reasonable to assume that global system constants were initialitzed with true constants (e.g. string literals), but user constants might have been initialized with something dynamically allocated, using a global initializer.

Conversion from retainable object pointer type in certain contexts

[beginning Apple 4.0, LLVM 3.1]

If an expression of retainable object pointer type is explicitly cast to a C retainable pointer type, the program is ill-formed as discussed above unless the result is immediately used:

- to initialize a parameter in an Objective-C message send where the parameter is not marked with the `cf_consumed` attribute, or
- to initialize a parameter in a direct call to an audited function where the parameter is not marked with the `cf_consumed` attribute.

Rationale

Consumed parameters are left out because ARC would naturally balance them with a retain, which was judged too treacherous. This is in part because several of the most common consuming functions are in the `Release` family, and it would be quite unfortunate for explicit releases to be silently balanced out in this way.

Ownership qualification

This section describes the behavior of *objects* of retainable object pointer type; that is, locations in memory which store retainable object pointers.

A type is a retainable object owner type if it is a retainable object pointer type or an array type whose element type is a retainable object owner type.

An ownership qualifier is a type qualifier which applies only to retainable object owner types. An array type is ownership-qualified according to its element type, and adding an ownership qualifier to an array type so qualifies its element type.

A program is ill-formed if it attempts to apply an ownership qualifier to a type which is already ownership-qualified, even if it is the same qualifier. There is a single exception to this rule: an ownership qualifier may be applied to a substituted template type parameter, which overrides the ownership qualifier provided by the template argument.

When forming a function type, the result type is adjusted so that any top-level ownership qualifier is deleted.

Except as described under the inference rules, a program is ill-formed if it attempts to form a pointer or reference type to a retainable object owner type which lacks an ownership qualifier.

Rationale

These rules, together with the inference rules, ensure that all objects and lvalues of retainable object pointer type have an ownership qualifier. The ability to override an ownership qualifier during template substitution is required to counteract the inference of __strong for template type arguments. Ownership qualifiers on return types are dropped because they serve no purpose there except to cause spurious problems with overloading and templates.

There are four ownership qualifiers:

- `__autoreleasing`
- `__strong`
- `__unsafe_unretained`
- `__weak`

A type is nontrivially ownership-qualified if it is qualified with `__autoreleasing`, `__strong`, or `__weak`.

Spelling

The names of the ownership qualifiers are reserved for the implementation. A program may not assume that they are or are not implemented with macros, or what those macros expand to.

An ownership qualifier may be written anywhere that any other type qualifier may be written.

If an ownership qualifier appears in the *declaration-specifiers*, the following rules apply:

- if the type specifier is a retainable object owner type, the qualifier initially applies to that type;
- otherwise, if the outermost non-array declarator is a pointer or block pointer declarator, the qualifier initially applies to that type;
- otherwise the program is ill-formed.
- If the qualifier is so applied at a position in the declaration where the next-innermost declarator is a function declarator, and there is an block declarator within that function declarator, then the qualifier applies instead to that block declarator and this rule is considered afresh beginning from the new position.

If an ownership qualifier appears on the declarator name, or on the declared object, it is applied to the innermost pointer or block-pointer type.

If an ownership qualifier appears anywhere else in a declarator, it applies to the type there.

Rationale

Ownership qualifiers are like `const` and `volatile` in the sense that they may sensibly apply at multiple distinct positions within a declarator. However, unlike those qualifiers, there are many situations where they are not meaningful, and so we make an effort to “move” the qualifier to a place where it will be meaningful. The general goal is to allow the programmer to write, say, `__strong` before the entire declaration and have it apply in the leftmost sensible place.

Property declarations

A property of retainable object pointer type may have ownership. If the property's type is ownership-qualified, then the property has that ownership. If the property has one of the following modifiers, then the property has the corresponding ownership. A property is ill-formed if it has conflicting sources of ownership, or if it has redundant ownership modifiers, or if it has `__autoreleasing` ownership.

- `assign` implies `__unsafe_unretained` ownership.
- `copy` implies `__strong` ownership, as well as the usual behavior of copy semantics on the setter.
- `retain` implies `__strong` ownership.
- `strong` implies `__strong` ownership.
- `unsafe_unretained` implies `__unsafe_unretained` ownership.
- `weak` implies `__weak` ownership.

With the exception of `weak`, these modifiers are available in non-ARC modes.

A property's specified ownership is preserved in its metadata, but otherwise the meaning is purely conventional unless the property is synthesized. If a property is synthesized, then the associated instance variable is the instance variable which is named, possibly implicitly, by the `@synthesize` declaration. If the associated instance variable already exists, then its ownership qualification must equal the ownership of the property; otherwise, the instance variable is created with that ownership qualification.

A property of retainable object pointer type which is synthesized without a source of ownership has the ownership of its associated instance variable, if it already exists; otherwise, [beginning Apple 3.1, LLVM 3.1] its ownership is implicitly `strong`. Prior to this revision, it was ill-formed to synthesize such a property.

Rationale

Using `strong` by default is safe and consistent with the generic ARC rule about inferring ownership. It is, unfortunately, inconsistent with the non-ARC rule which states that such properties are implicitly `assign`. However, that rule is clearly untenable in ARC, since it leads to default-unsafe code. The main merit to banning the properties is to avoid confusion with non-ARC practice, which did not ultimately strike us as sufficient to justify requiring extra syntax and (more importantly) forcing novices to understand ownership rules just to declare a property when the default is so reasonable. Changing the rule away from non-ARC practice was acceptable because we had conservatively banned the synthesis in order to give ourselves exactly this leeway.

Applying `__attribute__((NSObject))` to a property not of retainable object pointer type has the same behavior it does outside of ARC: it requires the property type to be some sort of pointer and permits the use of modifiers other than `assign`. These modifiers only affect the synthesized getter and setter; direct accesses to the ivar (even if synthesized) still have primitive semantics, and the value in the ivar will not be automatically released during deallocation.

Semantics

There are five managed operations which may be performed on an object of retainable object pointer type. Each qualifier specifies different semantics for each of these operations. It is still undefined behavior to access an object outside of its lifetime.

A load or store with "primitive semantics" has the same semantics as the respective operation would have on an `void*` lvalue with the same alignment and non-ownership qualification.

Reading occurs when performing a lvalue-to-rvalue conversion on an object lvalue.

- For `__weak` objects, the current pointee is retained and then released at the end of the current full-expression. This must execute atomically with respect to assignments and to the final release of the pointee.
- For all other objects, the lvalue is loaded with primitive semantics.

Assignment occurs when evaluating an assignment operator. The semantics vary based on the qualification:

- For `__strong` objects, the new pointee is first retained; second, the lvalue is loaded with primitive semantics; third, the new pointee is stored into the lvalue with primitive semantics; and finally, the old pointee is released. This is not performed atomically; external synchronization must be used to make this safe in the face of concurrent loads and stores.
- For `__weak` objects, the lvalue is updated to point to the new pointee, unless the new pointee is an object currently undergoing deallocation, in which case the lvalue is updated to a null pointer. This must execute atomically with respect to other assignments to the object, to reads from the object, and to the final release of the new pointee.
- For `__unsafe_unretained` objects, the new pointee is stored into the lvalue using primitive semantics.
- For `__autoreleasing` objects, the new pointee is retained, autoreleased, and stored into the lvalue using primitive semantics.

Initialization occurs when an object's lifetime begins, which depends on its storage duration. Initialization proceeds in two stages:

1. First, a null pointer is stored into the lvalue using primitive semantics. This step is skipped if the object is `__unsafe_unretained`.
2. Second, if the object has an initializer, that expression is evaluated and then assigned into the object using the usual assignment semantics.

Destruction occurs when an object's lifetime ends. In all cases it is semantically equivalent to assigning a null pointer to the object, with the proviso that of course the object cannot be legally read after the object's lifetime ends.

Moving occurs in specific situations where an lvalue is "moved from", meaning that its current pointee will be used but the object may be left in a different (but still valid) state. This arises with `__block` variables and rvalue references in C++. For `__strong` lvalues, moving is equivalent to loading the lvalue with primitive semantics, writing a null pointer to it with primitive semantics, and then releasing the result of the load at the end of the current full-expression. For all other lvalues, moving is equivalent to reading the object.

Restrictions

Weak-unavailable types

It is explicitly permitted for Objective-C classes to not support `__weak` references. It is undefined behavior to perform an operation with weak assignment semantics with a pointer to an Objective-C object whose class does not support `__weak` references.

Rationale

Historically, it has been possible for a class to provide its own reference-count implementation by overriding `retain`, `release`, etc. However, weak references to an object require coordination with its class's reference-count implementation because, among other things, weak loads and stores must be atomic with respect to the final release. Therefore, existing custom reference-count implementations will generally not support weak references without additional effort. This is unavoidable without breaking binary compatibility.

A class may indicate that it does not support weak references by providing the `objc_arc_weak_reference_unavailable` attribute on the class's interface declaration. A retainable object pointer type is **weak-unavailable** if is a pointer to an (optionally protocol-qualified) Objective-C class `T` where `T` or one of its superclasses has the `objc_arc_weak_reference_unavailable` attribute. A program is ill-formed if it applies the `__weak` ownership qualifier to a weak-unavailable type or if the value operand of a weak assignment operation has a weak-unavailable type.

Storage duration of `__autoreleasing` objects

A program is ill-formed if it declares an `__autoreleasing` object of non-automatic storage duration. A program is ill-formed if it captures an `__autoreleasing` object in a block or, unless by reference, in a C++11 lambda.

Rationale

Autorelease pools are tied to the current thread and scope by their nature. While it is possible to have temporary objects whose instance variables are filled with autoreleased objects, there is no way that ARC can provide any sort of safety guarantee there.

It is undefined behavior if a non-null pointer is assigned to an `__autoreleasing` object while an autorelease pool is in scope and then that object is read after the autorelease pool's scope is left.

Conversion of pointers to ownership-qualified types

A program is ill-formed if an expression of type `T*` is converted, explicitly or implicitly, to the type `U*`, where `T` and `U` have different ownership qualification, unless:

- `T` is qualified with `__strong`, `__autoreleasing`, or `__unsafe_unretained`, and `U` is qualified with both `const` and `__unsafe_unretained`; or
- either `T` or `U` is `cv void`, where `cv` is an optional sequence of non-ownership qualifiers; or
- the conversion is requested with a `reinterpret_cast` in Objective-C++; or
- the conversion is a well-formed pass-by-writeback.

The analogous rule applies to `T&` and `U&` in Objective-C++.

Rationale

These rules provide a reasonable level of type-safety for indirect pointers, as long as the underlying memory is not deallocated. The conversion to `const __unsafe_unretained` is permitted because the semantics of reads are equivalent across all these ownership semantics, and that's a very useful and common pattern. The interconversion with `void*` is useful for allocating memory or otherwise escaping the type system, but use it carefully. `reinterpret_cast` is considered to be an obvious enough sign of taking responsibility for any problems.

It is undefined behavior to access an ownership-qualified object through an lvalue of a differently-qualified type, except that any non-`__weak` object may be read through an `__unsafe_unretained` lvalue.

It is undefined behavior if the storage of a `__strong` or `__weak` object is not properly initialized before the first managed operation is performed on the object, or if the storage of such an object is freed or reused before the object has been properly deinitialized. Storage for a `__strong` or `__weak` object may be properly initialized by filling it with the representation of a null pointer, e.g. by acquiring the memory with `calloc` or using `bzero` to zero it out. A `__strong` or `__weak` object may be properly deinitialized by assigning a null pointer into it. A `__strong` object may also be properly initialized by copying into it (e.g. with `memcpy`) the representation of a different `__strong` object whose storage has been properly initialized; doing this properly deinitializes the source object and causes its storage to no longer be properly initialized. A `__weak` object may not be representation-copied in this way.

These requirements are followed automatically for objects whose initialization and deinitialization are under the control of ARC:

- objects of static, automatic, and temporary storage duration
- instance variables of Objective-C objects
- elements of arrays where the array object's initialization and deinitialization are under the control of ARC
- fields of Objective-C struct types where the struct object's initialization and deinitialization are under the control of ARC
- non-static data members of Objective-C++ non-union class types
- Objective-C++ objects and arrays of dynamic storage duration created with the `new` or `new[]` operators and destroyed with the corresponding `delete` or `delete[]` operator

They are not followed automatically for these objects:

- objects of dynamic storage duration created in other memory, such as that returned by `malloc`
- union members

Rationale

ARC must perform special operations when initializing an object and when destroying it. In many common situations, ARC knows when an object is created and when it is destroyed and can ensure that these operations are performed correctly. Otherwise, however, ARC requires programmer cooperation to establish its initialization invariants because it is infeasible for ARC to dynamically infer whether they are intact. For example, there is no syntactic difference in C between an assignment that is intended by the programmer to initialize a variable and one that is intended to replace the existing value stored there, but ARC must perform one operation or the other. ARC chooses to always assume that objects are initialized (except when it is in charge of initializing them) because the only workable alternative would be to ban all code patterns that could potentially be used to access uninitialized memory, and that would be too limiting. In practice, this is rarely a problem because programmers do not generally need to work with objects for which the requirements are not handled automatically.

Note that dynamically-allocated Objective-C++ arrays of nontrivially-ownership-qualified type are not ABI-compatible with non-ARC code because the non-ARC code will consider the element type to be POD. Such arrays that are `new[]`'d in ARC translation units cannot be `delete[]`'d in non-ARC translation units and vice-versa.

Passing to an out parameter by writeback

If the argument passed to a parameter of type `T __autoreleasing *` has type `U oq *`, where `oq` is an ownership qualifier, then the argument is a candidate for pass-by-writeback` if:

- `oq` is `__strong` or `__weak`, and
- it would be legal to initialize a `T __strong *` with a `U __strong *`.

For purposes of overload resolution, an implicit conversion sequence requiring a pass-by-writeback is always worse than an implicit conversion sequence not requiring a pass-by-writeback.

The pass-by-writeback is ill-formed if the argument expression does not have a legal form:

- `&var`, where `var` is a scalar variable of automatic storage duration with retainable object pointer type
- a conditional expression where the second and third operands are both legal forms
- a cast whose operand is a legal form
- a null pointer constant

Rationale

The restriction in the form of the argument serves two purposes. First, it makes it impossible to pass the address of an array to the argument, which serves to protect against an otherwise serious risk of mis-inferring an "array" argument as an out-parameter. Second, it makes it much less likely that the user will see confusing aliasing problems due to the implementation, below, where their store to the writeback temporary is not immediately seen in the original argument variable.

A pass-by-writeback is evaluated as follows:

1. The argument is evaluated to yield a pointer `p` of type `U oq *`.
2. If `p` is a null pointer, then a null pointer is passed as the argument, and no further work is required for the pass-by-writeback.
3. Otherwise, a temporary of type `T __autoreleasing` is created and initialized to a null pointer.
4. If the parameter is not an Objective-C method parameter marked `out`, then `*p` is read, and the result is written into the temporary with primitive semantics.
5. The address of the temporary is passed as the argument to the actual call.

6. After the call completes, the temporary is loaded with primitive semantics, and that value is assigned into `*p`.

Rationale

This is all admittedly convoluted. In an ideal world, we would see that a local variable is being passed to an out-parameter and retroactively modify its type to be `__autoreleasing` rather than `__strong`. This would be remarkably difficult and not always well-founded under the C type system. However, it was judged unacceptably invasive to require programmers to write `__autoreleasing` on all the variables they intend to use for out-parameters. This was the least bad solution.

Ownership-qualified fields of structs and unions

A member of a struct or union may be declared to have ownership-qualified type. If the type is qualified with `__unsafe_unretained`, the semantics of the containing aggregate are unchanged from the semantics of an unqualified type in a non-ARC mode. If the type is qualified with `__autoreleasing`, the program is ill-formed. Otherwise, if the type is nontrivially ownership-qualified, additional rules apply.

Both Objective-C and Objective-C++ support nontrivially ownership-qualified fields. Due to formal differences between the standards, the formal treatment is different; however, the basic language model is intended to be the same for identical code.

Rationale

Permitting `__strong` and `__weak` references in aggregate types allows programmers to take advantage of the normal language tools of C and C++ while still automatically managing memory. While it is usually simpler and more idiomatic to use Objective-C objects for secondary data structures, doing so can introduce extra allocation and message-send overhead, which can cause to unacceptable performance. Using structs can resolve some of this tension.

`__autoreleasing` is forbidden because it is treacherous to rely on autoreleases as an ownership tool outside of a function-local contexts.

Earlier releases of Clang permitted `__strong` and `__weak` only references in Objective-C++ classes, not in Objective-C. This restriction was an undesirable short-term constraint arising from the complexity of adding support for non-trivial struct types to C.

In Objective-C++, nontrivially ownership-qualified types are treated for nearly all purposes as if they were class types with non-trivial default constructors, copy constructors, move constructors, copy assignment operators, move assignment operators, and destructors. This includes the determination of the triviality of special members of classes with a non-static data member of such a type.

In Objective-C, the definition cannot be so succinct: because the C standard lacks rules for non-trivial types, those rules must first be developed. They are given in the next section. The intent is that these rules are largely consistent with the rules of C++ for code expressible in both languages.

Formal rules for non-trivial types in C

The following are base rules which can be added to C to support implementation-defined non-trivial types.

A type in C is said to be *non-trivial to copy*, *non-trivial to destroy*, or *non-trivial to default-initialize* if:

- it is a struct or union containing a member whose type is non-trivial to (respectively) copy, destroy, or default-initialize;
- it is a qualified type whose unqualified type is non-trivial to (respectively) copy, destroy, or default-initialize (for at least the standard C qualifiers); or
- it is an array type whose element type is non-trivial to (respectively) copy, destroy, or default-initialize.

A type in C is said to be *illegal to copy*, *illegal to destroy*, or *illegal to default-initialize* if:

- it is a union which contains a member whose type is either illegal or non-trivial to (respectively) copy, destroy, or initialize;
- it is a qualified type whose unqualified type is illegal to (respectively) copy, destroy, or default-initialize (for at least the standard C qualifiers); or
- it is an array type whose element type is illegal to (respectively) copy, destroy, or default-initialize.

No type describable under the rules of the C standard shall be either non-trivial or illegal to copy, destroy, or default-initialize. An implementation may provide additional types which have one or more of these properties.

An expression calls for a type to be copied if it:

- passes an argument of that type to a function call,
- defines a function which declares a parameter of that type,
- calls or defines a function which returns a value of that type,
- assigns to an l-value of that type, or
- converts an l-value of that type to an r-value.

A program calls for a type to be destroyed if it:

- passes an argument of that type to a function call,
- defines a function which declares a parameter of that type,
- calls or defines a function which returns a value of that type,
- creates an object of automatic storage duration of that type,
- assigns to an l-value of that type, or
- converts an l-value of that type to an r-value.

A program calls for a type to be default-initialized if it:

- declares a variable of that type without an initializer.

An expression is ill-formed if calls for a type to be copied, destroyed, or default-initialized and that type is illegal to (respectively) copy, destroy, or default-initialize.

A program is ill-formed if it contains a function type specifier with a parameter or return type that is illegal to copy or destroy. If a function type specifier would be ill-formed for this reason except that the parameter or return type was incomplete at that point in the translation unit, the program is ill-formed but no diagnostic is required.

A `goto` or `switch` is ill-formed if it jumps into the scope of an object of automatic storage duration whose type is non-trivial to destroy.

C specifies that it is generally undefined behavior to access an l-value if there is no object of that type at that location. Implementations are often lenient about this, but non-trivial types generally require it to be enforced more strictly. The following rules apply:

The *static subobjects* of a type `T` at a location `L` are:

- an object of type `T` spanning from `L` to `L + sizeof(T)`;
- if `T` is a struct type, then for each field `f` of that struct, the static subobjects of `T` at location `L + offsetof(T, .f)`; and
- if `T` is the array type `E[N]`, then for each `i` satisfying `0 <= i < N`, the static subobjects of `E` at location `L + i * sizeof(E)`.

If an l-value is converted to an r-value, then all static subobjects whose types are non-trivial to copy are accessed. If an l-value is assigned to, or if an object of automatic storage duration goes out of scope, then all static subobjects of types that are non-trivial to destroy are accessed.

A dynamic object is created at a location if an initialization initializes an object of that type there. A dynamic object ceases to exist at a location if the memory is repurposed. Memory is repurposed if it is freed or if a different dynamic object is created there, for example by assigning into a different union member. An implementation may provide additional rules for what constitutes creating or destroying a dynamic object.

If an object is accessed under these rules at a location where no such dynamic object exists, the program has undefined behavior. If memory for a location is repurposed while a dynamic object that is non-trivial to destroy exists at that location, the program has undefined behavior.

Rationale

While these rules are far less fine-grained than C++, they are nonetheless sufficient to express a wide spectrum of types. Types that express some sort of ownership will generally be non-trivial to both copy and destroy and either non-trivial or illegal to default-initialize. Types that don't express ownership may still be non-trivial to copy because of some sort of address sensitivity; for example, a relative reference. Distinguishing default initialization allows types to impose policies about how they are created.

These rules assume that assignment into an l-value is always a modification of an existing object rather than an initialization. Assignment is then a compound operation where the old value is read and destroyed, if necessary, and the new value is put into place. These are the natural semantics of value propagation, where all basic operations on the type come down to copies and destroys, and everything else is just an optimization on top of those.

The most glaring weakness of programming with non-trivial types in C is that there are no language mechanisms (akin to C++'s placement `new` and explicit destructor calls) for explicitly creating and destroying objects. Clang should consider adding builtins for this purpose, as well as for common optimizations like destructive relocation.

Application of the formal C rules to nontrivial ownership qualifiers

Nontrivially ownership-qualified types are considered non-trivial to copy, destroy, and default-initialize.

A dynamic object of nontrivially ownership-qualified type contingently exists at a location if the memory is filled with a zero pattern, e.g. by `calloc` or `bzero`. Such an object can be safely accessed in all of the cases above, but its memory can also be safely repurposed. Assigning a null pointer into an l-value of `__weak` or `__strong`-qualified type accesses the dynamic object there (and thus may have undefined behavior if no such object exists), but afterwards the object's memory is guaranteed to be filled with a zero pattern and thus may be either further accessed or repurposed as needed. The upshot is that programs may safely initialize dynamically-allocated memory for nontrivially ownership-qualified types by ensuring it is zero-initialized, and they may safely deinitialize memory before freeing it by storing `nil` into any `__strong` or `__weak` references previously created in that memory.

C/C++ compatibility for structs and unions with non-trivial members

Structs and unions with non-trivial members are compatible in different language modes (e.g. between Objective-C and Objective-C++, or between ARC and non-ARC modes) under the following conditions:

- The types must be compatible ignoring ownership qualifiers according to the baseline, non-ARC rules (e.g. C struct compatibility or C++'s ODR). This condition implies a pairwise correspondance between fields.

  Note that an Objective-C++ class with base classes, a user-provided copy or move constructor, or a user-provided destructor is never compatible with an Objective-C type.

- If two fields correspond as above, and at least one of the fields is ownership-qualified, then:

  - the fields must be identically qualified, or else
  - one type must be unqualified (and thus declared in a non-ARC mode), and the other type must be qualified with `__unsafe_unretained` or `__strong`.

  Note that `__weak` fields must always be declared `__weak` because of the need to pin those fields in memory and keep them properly registered with the Objective-C runtime. Non-ARC modes may still declare fields `__weak` by enabling `-fobjc-weak`.

These compatibility rules permit a function that takes a parameter of non-trivial struct type to be written in ARC and called from non-ARC or vice-versa. The convention for this always transfers ownership of objects stored in `__strong` fields from the caller to the callee, just as for an `ns_consumed` argument. Therefore, non-ARC callers must ensure that such fields are initialized to a +1 reference, and non-ARC callees must balance that +1 by releasing the reference or transferring it as appropriate.

Likewise, a function returning a non-trivial struct may be written in ARC and called from non-ARC or vice-versa. The convention for this always transfers ownership of objects stored in `__strong` fields from the callee to the caller, and so callees must initialize such fields with +1 references, and callers must balance that +1 by releasing or transferring them.

Similar transfers of responsibility occur for `__weak` fields, but since both sides must use native `__weak` support to ensure calling convention compatibility, this transfer is always handled automatically by the compiler.

Rationale

In earlier releases, when non-trivial ownership was only permitted on fields in Objective-C++, the ABI used for such classees was the ordinary ABI for non-trivial C++ classes, which passes arguments and returns indirectly and does not transfer responsibility for arguments. When support for Objective-C structs was added, it was decided to change to the current ABI for three reasons:

- It permits ARC / non-ARC compatibility for structs containing only `__strong` references, as long as the non-ARC side is careful about transferring ownership.
- It avoids unnecessary indirection for sufficiently small types that the C ABI would prefer to pass in registers.
- Given that struct arguments must be produced at +1 to satisfy C's semantics of initializing the local parameter variable, transferring ownership of that copy to the callee is generally better for ARC optimization, since otherwise there will be releases in the caller that are much harder to pair with transfers in the callee.

Breaking compatibility with existing Objective-C++ structures was considered an acceptable cost, as most Objective-C++ code does not have binary-compatibility requirements. Any existing code which cannot accept this compatibility break, which is necessarily Objective-C++, should force the use of the standard C++ ABI by declaring an empty (but non-defaulted) destructor.

Ownership inference

Objects

If an object is declared with retainable object owner type, but without an explicit ownership qualifier, its type is implicitly adjusted to have `__strong` qualification.

As a special case, if the object's base type is `Class` (possibly protocol-qualified), the type is adjusted to have `__unsafe_unretained` qualification instead.

Indirect parameters

If a function or method parameter has type `T*`, where `T` is an ownership-unqualified retainable object pointer type, then:

- if `T` is `const`-qualified or `Class`, then it is implicitly qualified with `__unsafe_unretained`;
- otherwise, it is implicitly qualified with `__autoreleasing`.

Rationale

`__autoreleasing` exists mostly for this case, the Cocoa convention for out-parameters. Since a pointer to `const` is obviously not an out-parameter, we instead use a type more useful for passing arrays. If the user instead intends to pass in a *mutable* array, inferring `__autoreleasing` is the wrong thing to do; this directs some of the caution in the following rules about writeback.

Such a type written anywhere else would be ill-formed by the general rule requiring ownership qualifiers.

This rule does not apply in Objective-C++ if a parameter's type is dependent in a template pattern and is only *instantiated* to a type which would be a pointer to an unqualified retainable object pointer type. Such code is still ill-formed.

Rationale

The convention is very unlikely to be intentional in template code.

Template arguments

If a template argument for a template type parameter is an retainable object owner type that does not have an explicit ownership qualifier, it is adjusted to have `__strong` qualification. This adjustment occurs regardless of whether the template argument was deduced or explicitly specified.

Rationale

`__strong` is a useful default for containers (e.g., `std::vector<id>`), which would otherwise require explicit qualification. Moreover, unqualified retainable object pointer types are unlikely to be useful within templates, since they generally need to have a qualifier applied to the before being used.

Method families

An Objective-C method may fall into a method family, which is a conventional set of behaviors ascribed to it by the Cocoa conventions.

A method is in a certain method family if:

- it has a `objc_method_family` attribute placing it in that family; or if not that,
- it does not have an `objc_method_family` attribute placing it in a different or no family, and
- its selector falls into the corresponding selector family, and
- its signature obeys the added restrictions of the method family.

A selector is in a certain selector family if, ignoring any leading underscores, the first component of the selector either consists entirely of the name of the method family or it begins with that name followed by a character other than a lowercase letter. For example, `_perform:with:` and `performWith:` would fall into the `perform` family (if we recognized one), but `performing:with` would not.

The families and their added restrictions are:

- `alloc` methods must return a retainable object pointer type.
- `copy` methods must return a retainable object pointer type.
- `mutableCopy` methods must return a retainable object pointer type.
- `new` methods must return a retainable object pointer type.
- `init` methods must be instance methods and must return an Objective-C pointer type. Additionally, a program is ill-formed if it declares or contains a call to an `init` method whose return type is neither `id` nor a pointer to a super-class or sub-class of the declaring class (if the method was declared on a class) or the static receiver type of the call (if it was declared on a protocol).

Rationale

There are a fair number of existing methods with `init`-like selectors which nonetheless don't follow the `init` conventions. Typically these are either accidental naming collisions or helper methods called during initialization. Because of the peculiar retain/release behavior of `init` methods, it's very important not to treat these methods as `init` methods if they aren't meant to be. It was felt that implicitly defining these methods out of the family based on the exact relationship between the return type and the declaring class would be much too subtle and fragile. Therefore we identify a small number of legitimate-seeming return types and call everything else an error. This serves the secondary purpose of encouraging programmers not to accidentally give methods names in the `init` family.

> Note that a method with an `init`-family selector which returns a non-Objective-C type (e.g. `void`) is perfectly well-formed; it simply isn't in the `init` family.

A program is ill-formed if a method's declarations, implementations, and overrides do not all have the same method family.

Explicit method family control

A method may be annotated with the `objc_method_family` attribute to precisely control which method family it belongs to. If a method in an `@implementation` does not have this attribute, but there is a method declared in the corresponding `@interface` that does, then the attribute is copied to the declaration in the `@implementation`. The attribute is available outside of ARC, and may be tested for with the preprocessor query `__has_attribute(objc_method_family)`.

The attribute is spelled `__attribute__((objc_method_family(` *family* `)))`. If *family* is `none`, the method has no family, even if it would otherwise be considered to have one based on its selector and type. Otherwise, *family* must be one of `alloc`, `copy`, `init`, `mutableCopy`, or `new`, in which case the method is considered to belong to the corresponding family regardless of its selector. It is an error if a method that is explicitly added to a family in this way does not meet the requirements of the family other than the selector naming convention.

Rationale

The rules codified in this document describe the standard conventions of Objective-C. However, as these conventions have not heretofore been enforced by an unforgiving mechanical system, they are only imperfectly kept, especially as they haven't always even been precisely defined. While it is possible to define low-level ownership semantics with attributes like `ns_returns_retained`, this attribute allows the user to communicate semantic intent, which is of use both to ARC (which, e.g., treats calls to `init` specially) and the static analyzer.

Semantics of method families

A method's membership in a method family may imply non-standard semantics for its parameters and return type.

Methods in the `alloc`, `copy`, `mutableCopy`, and `new` families — that is, methods in all the currently-defined families except `init` — implicitly return a retained object as if they were annotated with the `ns_returns_retained` attribute. This can be overridden by annotating the method with either of the `ns_returns_autoreleased` or `ns_returns_not_retained` attributes.

Properties also follow same naming rules as methods. This means that those in the `alloc`, `copy`, `mutableCopy`, and `new` families provide access to retained objects. This can be overridden by annotating the property with `ns_returns_not_retained` attribute.

Semantics of `init`

Methods in the `init` family implicitly consume their `self` parameter and return a retained object. Neither of these properties can be altered through attributes.

A call to an `init` method with a receiver that is either `self` (possibly parenthesized or casted) or `super` is called a delegate init call. It is an error for a delegate init call to be made except from an `init` method, and excluding blocks within such methods.

As an exception to the usual rule, the variable `self` is mutable in an `init` method and has the usual semantics for a `__strong` variable. However, it is undefined behavior and the program is ill-formed, no diagnostic required, if an `init` method attempts to use the previous value of `self` after the completion of a delegate init call. It is conventional, but not required, for an `init` method to return `self`.

It is undefined behavior for a program to cause two or more calls to `init` methods on the same object, except that each `init` method invocation may perform at most one delegate init call.

Related result types

Certain methods are candidates to have related result types:

- class methods in the `alloc` and `new` method families
- instance methods in the `init` family
- the instance method `self`
- outside of ARC, the instance methods `retain` and `autorelease`

If the formal result type of such a method is `id` or protocol-qualified `id`, or a type equal to the declaring class or a superclass, then it is said to have a related result type. In this case, when invoked in an explicit message send, it is assumed to return a type related to the type of the receiver:

- if it is a class method, and the receiver is a class name `T`, the message send expression has type `T*`; otherwise
- if it is an instance method, and the receiver has type `T`, the message send expression has type `T`; otherwise
- the message send expression has the normal result type of the method.

This is a new rule of the Objective-C language and applies outside of ARC.

Rationale

ARC's automatic code emission is more prone than most code to signature errors, i.e. errors where a call was emitted against one method signature, but the implementing method has an incompatible signature. Having more precise type information helps drastically lower this risk, as well as catching a number of latent bugs.

Optimization

Within this section, the word function will be used to refer to any structured unit of code, be it a C function, an Objective-C method, or a block.

This specification describes ARC as performing specific `retain` and `release` operations on retainable object pointers at specific points during the execution of a program. These operations make up a non-contiguous subsequence of the computation history of the program. The portion of this sequence for a particular retainable object pointer for which a specific function execution is directly responsible is the formal local retain history of the object pointer. The corresponding actual sequence executed is the *dynamic local retain history*.

However, under certain circumstances, ARC is permitted to re-order and eliminate operations in a manner which may alter the overall computation history beyond what is permitted by the general "as if" rule of C/C++ and the restrictions on the implementation of `retain` and `release`.

Rationale

Specifically, ARC is sometimes permitted to optimize `release` operations in ways which might cause an object to be deallocated before it would otherwise be. Without this, it would be almost impossible to eliminate any `retain`/`release` pairs. For example, consider the following code:

```
id x = _ivar;
[x foo];
```

If we were not permitted in any event to shorten the lifetime of the object in `x`, then we would not be able to eliminate this retain and release unless we could prove that the message send could not modify `_ivar` (or deallocate `self`). Since message sends are opaque to the optimizer, this is not possible, and so ARC's hands would be almost completely tied.

ARC makes no guarantees about the execution of a computation history which contains undefined behavior. In particular, ARC makes no guarantees in the presence of race conditions.

ARC may assume that any retainable object pointers it receives or generates are instantaneously valid from that point until a point which, by the concurrency model of the host language, happens-after the generation of the pointer and happens-before a release of that object (possibly via an aliasing pointer or indirectly due to destruction of a different object).

Rationale

There is very little point in trying to guarantee correctness in the presence of race conditions. ARC does not have a stack-scanning garbage collector, and guaranteeing the atomicity of every load and store operation would be prohibitive and preclude a vast amount of optimization.

ARC may assume that non-ARC code engages in sensible balancing behavior and does not rely on exact or minimum retain count values except as guaranteed by `__strong` object invariants or +1 transfer conventions. For example, if an object is provably double-retained and double-released, ARC may eliminate the inner retain and release; it does not need to guard against code which performs an unbalanced release followed by a "balancing" retain.

Object liveness

ARC may not allow a retainable object `X` to be deallocated at a time `T` in a computation history if:

- `X` is the value stored in a `__strong` object `S` with precise lifetime semantics, or
- `X` is the value stored in a `__strong` object `S` with imprecise lifetime semantics and, at some point after `T` but before the next store to `S`, the computation history features a load from `S` and in some way depends on the value loaded, or
- `X` is a value described as being released at the end of the current full-expression and, at some point after `T` but before the end of the full-expression, the computation history depends on that value.

Rationale

The intent of the second rule is to say that objects held in normal `__strong` local variables may be released as soon as the value in the variable is no longer being used: either the variable stops being used completely or a new value is stored in the variable.

The intent of the third rule is to say that return values may be released after they've been used.

A computation history depends on a pointer value `P` if it:

- performs a pointer comparison with `P`,
- loads from `P`,
- stores to `P`,
- depends on a pointer value `Q` derived via pointer arithmetic from `P` (including an instance-variable or field access), or
- depends on a pointer value `Q` loaded from `P`.

Dependency applies only to values derived directly or indirectly from a particular expression result and does not occur merely because a separate pointer value dynamically aliases `P`. Furthermore, this dependency is not carried by values that are stored to objects.

Rationale

The restrictions on dependency are intended to make this analysis feasible by an optimizer with only incomplete information about a program. Essentially, dependence is carried to "obvious" uses of a pointer. Merely passing a pointer argument to a function does not itself cause dependence, but since generally the optimizer will not be

able to prove that the function doesn't depend on that parameter, it will be forced to conservatively assume it does.

Dependency propagates to values loaded from a pointer because those values might be invalidated by deallocating the object. For example, given the code `__strong id x = p->ivar;`, ARC must not move the release of `p` to between the load of `p->ivar` and the retain of that value for storing into `x`.

Dependency does not propagate through stores of dependent pointer values because doing so would allow dependency to outlive the full-expression which produced the original value. For example, the address of an instance variable could be written to some global location and then freely accessed during the lifetime of the local, or a function could return an inner pointer of an object and store it to a local. These cases would be potentially impossible to reason about and so would basically prevent any optimizations based on imprecise lifetime. There are also uncommon enough to make it reasonable to require the precise-lifetime annotation if someone really wants to rely on them.

Dependency does propagate through return values of pointer type. The compelling source of need for this rule is a property accessor which returns an un-autoreleased result; the calling function must have the chance to operate on the value, e.g. to retain it, before ARC releases the original pointer. Note again, however, that dependence does not survive a store, so ARC does not guarantee the continued validity of the return value past the end of the full-expression.

No object lifetime extension

If, in the formal computation history of the program, an object `X` has been deallocated by the time of an observable side-effect, then ARC must cause `X` to be deallocated by no later than the occurrence of that side-effect, except as influenced by the re-ordering of the destruction of objects.

Rationale

This rule is intended to prohibit ARC from observably extending the lifetime of a retainable object, other than as specified in this document. Together with the rule limiting the transformation of releases, this rule requires ARC to eliminate retains and release only in pairs.

ARC's power to reorder the destruction of objects is critical to its ability to do any optimization, for essentially the same reason that it must retain the power to decrease the lifetime of an object. Unfortunately, while it's generally poor style for the destruction of objects to have arbitrary side-effects, it's certainly possible. Hence the caveat.

Precise lifetime semantics

In general, ARC maintains an invariant that a retainable object pointer held in a `__strong` object will be retained for the full formal lifetime of the object. Objects subject to this invariant have precise lifetime semantics.

By default, local variables of automatic storage duration do not have precise lifetime semantics. Such objects are simply strong references which hold values of retainable object pointer type, and these values are still fully subject to the optimizations on values under local control.

Rationale

Applying these precise-lifetime semantics strictly would be prohibitive. Many useful optimizations that might theoretically decrease the lifetime of an object would be rendered impossible. Essentially, it promises too much.

A local variable of retainable object owner type and automatic storage duration may be annotated with the `objc_precise_lifetime` attribute to indicate that it should be considered to be an object with precise lifetime semantics.

Rationale

Nonetheless, it is sometimes useful to be able to force an object to be released at a precise time, even if that object does not appear to be used. This is likely to be uncommon enough that the syntactic weight of explicitly requesting these semantics will not be burdensome, and may even make the code clearer.

Miscellaneous

Special methods

Memory management methods

A program is ill-formed if it contains a method definition, message send, or `@selector` expression for any of the following selectors:

- `autorelease`
- `release`
- `retain`
- `retainCount`

Rationale

`retainCount` is banned because ARC robs it of consistent semantics. The others were banned after weighing three options for how to deal with message sends:

Honoring them would work out very poorly if a programmer naively or accidentally tried to incorporate code written for manual retain/release code into an ARC program. At best, such code would do twice as much work as necessary; quite frequently, however, ARC and the explicit code would both try to balance the same retain, leading to crashes. The cost is losing the ability to perform "unrooted" retains, i.e. retains not logically corresponding to a strong reference in the object graph.

Ignoring them would badly violate user expectations about their code. While it *would* make it easier to develop code simultaneously for ARC and non-ARC, there is very little reason to do so except for certain library developers. ARC and non-ARC translation units share an execution model and can seamlessly interoperate. Within a translation unit, a developer who faithfully maintains their code in non-ARC mode is suffering all the restrictions of ARC for zero benefit, while a developer who isn't testing the non-ARC mode is likely to be unpleasantly surprised if they try to go back to it.

Banning them has the disadvantage of making it very awkward to migrate existing code to ARC. The best answer to that, given a number of other changes and restrictions in ARC, is to provide a specialized tool to assist users in that migration.

Implementing these methods was banned because they are too integral to the semantics of ARC; many tricks which worked tolerably under manual reference counting will misbehave if ARC performs an ephemeral extra retain or two. If absolutely required, it is still possible to implement them in non-ARC code, for example in a category; the implementations must obey the semantics laid out elsewhere in this document.

`dealloc`

A program is ill-formed if it contains a message send or `@selector` expression for the selector `dealloc`.

Rationale

There are no legitimate reasons to call `dealloc` directly.

A class may provide a method definition for an instance method named `dealloc`. This method will be called after the final `release` of the object but before it is deallocated or any of its instance variables are destroyed. The superclass's implementation of `dealloc` will be called automatically when the method returns.

Rationale

Even though ARC destroys instance variables automatically, there are still legitimate reasons to write a `dealloc` method, such as freeing non-retainable resources. Failing to call `[super dealloc]` in such a method is nearly always a bug. Sometimes, the object is simply trying to prevent itself from being destroyed, but `dealloc` is really far too late for the object to be raising such objections. Somewhat more legitimately, an object may have been pool-allocated and should not be deallocated with `free`; for now, this can only be supported with a `dealloc` implementation outside of ARC. Such an implementation must be very careful to do all the other work that `NSObject`'s `dealloc` would, which is outside the scope of this document to describe.

The instance variables for an ARC-compiled class will be destroyed at some point after control enters the `dealloc` method for the root class of the class. The ordering of the destruction of instance variables is unspecified, both within a single class and between subclasses and superclasses.

Rationale

The traditional, non-ARC pattern for destroying instance variables is to destroy them immediately before calling `[super dealloc]`. Unfortunately, message sends from the superclass are quite capable of reaching methods in the subclass, and those methods may well read or write to those instance variables. Making such message sends from dealloc is generally discouraged, since the subclass may well rely on other invariants that were broken during `dealloc`, but it's not so inescapably dangerous that we felt comfortable calling it undefined behavior. Therefore we chose to delay destroying the instance variables to a point at which message sends are clearly disallowed: the point at which the root class's deallocation routines take over.

In most code, the difference is not observable. It can, however, be observed if an instance variable holds a strong reference to an object whose deallocation will trigger a side-effect which must be carefully ordered with respect to the destruction of the super class. Such code violates the design principle that semantically important behavior should be explicit. A simple fix is to clear the instance variable manually during `dealloc`; a more holistic solution is to move semantically important side-effects out of `dealloc` and into a separate teardown phase which can rely on working with well-formed objects.

`@autoreleasepool`

To simplify the use of autorelease pools, and to bring them under the control of the compiler, a new kind of statement is available in Objective-C. It is written `@autoreleasepool` followed by a *compound-statement*, i.e. by a new scope delimited by curly braces. Upon entry to this block, the current state of the autorelease pool is captured. When the block is exited normally, whether by fallthrough or directed control flow (such as `return` or `break`), the autorelease pool is restored to the saved state, releasing all the objects in it. When the block is exited with an exception, the pool is not drained.

`@autoreleasepool` may be used in non-ARC translation units, with equivalent semantics.

A program is ill-formed if it refers to the `NSAutoreleasePool` class.

Rationale

Autorelease pools are clearly important for the compiler to reason about, but it is far too much to expect the compiler to accurately reason about control dependencies between two calls. It is also very easy to accidentally forget to drain an autorelease pool when using the manual API, and this can significantly inflate the process's high-water-mark. The introduction of a new scope is unfortunate but basically required for sane interaction with the rest of the language. Not draining the pool during an unwind is apparently required by the Objective-C exceptions implementation.

Externally-Retained Variables

In some situations, variables with strong ownership are considered externally-retained by the implementation. This means that the variable is retained elsewhere, and therefore the implementation can elide retaining and releasing its value. Such a variable is implicitly `const` for safety. In contrast with `__unsafe_unretained`, an externally-retained variable still behaves as a strong variable outside of initialization and destruction. For instance, when an externally-retained variable is captured in a block the value of the variable is retained and released on block capture and destruction. It also affects C++ features such as lambda capture, `decltype`, and template argument deduction.

Implicitly, the implementation assumes that the self parameter in a non-init method and the variable in a for-in loop are externally-retained.

Externally-retained semantics can also be opted into with the `objc_externally_retained` attribute. This attribute can apply to strong local variables, functions, methods, or blocks:

```
@class WobbleAmount;

@interface Widget : NSObject
-(void)wobble:(WobbleAmount *)amount;
@end

@implementation Widget

-(void)wobble:(WobbleAmount *)amount
         __attribute__((objc_externally_retained)) {
  // 'amount' and 'alias' aren't retained on entry, nor released on exit.
  __attribute__((objc_externally_retained)) WobbleAmount *alias = amount;
}
@end
```

Annotating a function with this attribute makes every parameter with strong retainable object pointer type externally-retained, unless the variable was explicitly qualified with `__strong`. For instance, `first_param` is externally-retained (and therefore `const`) below, but not `second_param`:

```
__attribute__((objc_externally_retained))
void f(NSArray *first_param, __strong NSArray *second_param) {
  // ...
}
```

You can test if your compiler has support for `objc_externally_retained` with `__has_attribute`:

```
#if __has_attribute(objc_externally_retained)
// Use externally retained...
#endif
```

`self`

The `self` parameter variable of an non-init Objective-C method is considered externally-retained by the implementation. It is undefined behavior, or at least dangerous, to cause an object to be deallocated during a message send to that object. In an init method, `self` follows the :ref:`init family rules <arc.family.semantics.init>`.

Rationale

The cost of retaining `self` in all methods was found to be prohibitive, as it tends to be live across calls, preventing the optimizer from proving that the retain and release are unnecessary — for good reason, as it's quite possible in theory to cause an object to be deallocated during its execution without this retain and release. Since it's extremely uncommon to actually do so, even unintentionally, and since there's no natural way for the programmer to remove this retain/release pair otherwise (as there is for other parameters by, say, making the variable `objc_externally_retained` or qualifying it with `__unsafe_unretained`), we chose to make this optimizing assumption and shift some amount of risk to the user.

Fast enumeration iteration variables

If a variable is declared in the condition of an Objective-C fast enumeration loop, and the variable has no explicit ownership qualifier, then it is implicitly externally-retained so that objects encountered during the enumeration are not actually retained and released.

Rationale

This is an optimization made possible because fast enumeration loops promise to keep the objects retained during enumeration, and the collection itself cannot be synchronously modified. It can be overridden by explicitly qualifying the variable with `__strong`, which will make the variable mutable again and cause the loop to retain the objects it encounters.

Blocks

The implicit `const` capture variables created when evaluating a block literal expression have the same ownership semantics as the local variables they capture. The capture is performed by reading from the captured variable and initializing the capture variable with that value; the capture variable is destroyed when the block literal is, i.e. at the end of the enclosing scope.

The inference rules apply equally to `__block` variables, which is a shift in semantics from non-ARC, where `__block` variables did not implicitly retain during capture.

`__block` variables of retainable object owner type are moved off the stack by initializing the heap copy with the result of moving from the stack copy.

With the exception of retains done as part of initializing a `__strong` parameter variable or reading a `__weak` variable, whenever these semantics call for retaining a value of block-pointer type, it has the effect of a `Block_copy`. The optimizer may remove such copies when it sees that the result is used only as an argument to a call.

When a block pointer type is converted to a non-block pointer type (such as `id`), `Block_copy` is called. This is necessary because a block allocated on the stack won't get copied to the heap when the non-block pointer escapes. A block pointer is implicitly converted to `id` when it is passed to a function as a variadic argument.

Exceptions

By default in Objective C, ARC is not exception-safe for normal releases:

- It does not end the lifetime of `__strong` variables when their scopes are abnormally terminated by an exception.
- It does not perform releases which would occur at the end of a full-expression if that full-expression throws an exception.

A program may be compiled with the option `-fobjc-arc-exceptions` in order to enable these, or with the option `-fno-objc-arc-exceptions` to explicitly disable them, with the last such argument "winning".

Rationale

The standard Cocoa convention is that exceptions signal programmer error and are not intended to be recovered from. Making code exceptions-safe by default would impose severe runtime and code size penalties on code that typically does not actually care about exceptions safety. Therefore, ARC-generated code leaks by default on exceptions, which is just fine if the process is going to be immediately terminated anyway. Programs which do care about recovering from exceptions should enable the option.

In Objective-C++, `-fobjc-arc-exceptions` is enabled by default.

Rationale

C++ already introduces pervasive exceptions-cleanup code of the sort that ARC introduces. C++ programmers who have not already disabled exceptions are much more likely to actual require exception-safety.

ARC does end the lifetimes of `__weak` objects when an exception terminates their scope unless exceptions are disabled in the compiler.

Rationale

The consequence of a local `__weak` object not being destroyed is very likely to be corruption of the Objective-C runtime, so we want to be safer here. Of course, potentially massive leaks are about as likely to take down the process as this corruption is if the program does try to recover from exceptions.

Interior pointers

An Objective-C method returning a non-retainable pointer may be annotated with the `objc_returns_inner_pointer` attribute to indicate that it returns a handle to the internal data of an object, and that this reference will be invalidated if the object is destroyed. When such a message is sent to an object, the object's lifetime will be extended until at least the earliest of:

- the last use of the returned pointer, or any pointer derived from it, in the calling function or
- the autorelease pool is restored to a previous state.

Rationale

Rationale: not all memory and resources are managed with reference counts; it is common for objects to manage private resources in their own, private way. Typically these resources are completely encapsulated within the object, but some classes offer their users direct access for efficiency. If ARC is not aware of methods that return such "interior" pointers, its optimizations can cause the owning object to be reclaimed too soon. This attribute informs ARC that it must tread lightly.

The extension rules are somewhat intentionally vague. The autorelease pool limit is there to permit a simple implementation to simply retain and autorelease the receiver. The other limit permits some amount of optimization. The phrase "derived from" is intended to encompass the results both of pointer transformations, such as casts and arithmetic, and of loading from such derived pointers; furthermore, it applies whether or not such derivations are applied directly in the calling code or by other utility code (for example, the C library routine `strchr`). However, the implementation never need account for uses after a return from the code which calls the method returning an interior pointer.

As an exception, no extension is required if the receiver is loaded directly from a `__strong` object with precise lifetime semantics.

Rationale

Implicit autoreleases carry the risk of significantly inflating memory use, so it's important to provide users a way of avoiding these autoreleases. Tying this to precise lifetime semantics is ideal, as for local variables this requires a very explicit annotation, which allows ARC to trust the user with good cheer.

C retainable pointer types

A type is a C retainable pointer type if it is a pointer to (possibly qualified) `void` or a pointer to a (possibly qualifier) `struct` or `class` type.

Rationale

ARC does not manage pointers of CoreFoundation type (or any of the related families of retainable C pointers which interoperate with Objective-C for retain/release operation). In fact, ARC does not even know how to distinguish these types from arbitrary C pointer types. The intent of this concept is to filter out some obviously non-object types while leaving a hook for later tightening if a means of exhaustively marking CF types is made available.

Auditing of C retainable pointer interfaces

[beginning Apple 4.0, LLVM 3.1]

A C function may be marked with the `cf_audited_transfer` attribute to express that, except as otherwise marked with attributes, it obeys the parameter (consuming vs. non-consuming) and return (retained vs. non-retained) conventions for a C function of its name, namely:

- A parameter of C retainable pointer type is assumed to not be consumed unless it is marked with the `cf_consumed` attribute, and
- A result of C retainable pointer type is assumed to not be returned retained unless the function is either marked `cf_returns_retained` or it follows the create/copy naming convention and is not marked `cf_returns_not_retained`.

A function obeys the create/copy naming convention if its name contains as a substring:

- either “Create” or “Copy” not followed by a lowercase letter, or
- either “create” or “copy” not followed by a lowercase letter and not preceded by any letter, whether uppercase or lowercase.

A second attribute, `cf_unknown_transfer`, signifies that a function’s transfer semantics cannot be accurately captured using any of these annotations. A program is ill-formed if it annotates the same function with both `cf_audited_transfer` and `cf_unknown_transfer`.

A pragma is provided to facilitate the mass annotation of interfaces:

```
#pragma clang arc_cf_code_audited begin
...
#pragma clang arc_cf_code_audited end
```

All C functions declared within the extent of this pragma are treated as if annotated with the `cf_audited_transfer` attribute unless they otherwise have the `cf_unknown_transfer` attribute. The pragma is accepted in all language modes. A program is ill-formed if it attempts to change files, whether by including a file or ending the current file, within the extent of this pragma.

It is possible to test for all the features in this section with `__has_feature(arc_cf_code_audited)`.

Rationale

A significant inconvenience in ARC programming is the necessity of interacting with APIs based around C retainable pointers. These features are designed to make it relatively easy for API authors to quickly review and annotate their interfaces, in turn improving the fidelity of tools such as the static analyzer and ARC. The single-file restriction on the pragma is designed to eliminate the risk of accidentally annotating some other header’s interfaces.

Runtime support

This section describes the interaction between the ARC runtime and the code generated by the ARC compiler. This is not part of the ARC language specification; instead, it is effectively a language-specific ABI supplement, akin to the “Itanium” generic ABI for C++.

Ownership qualification does not alter the storage requirements for objects, except that it is undefined behavior if a `__weak` object is inadequately aligned for an object of type `id`. The other qualifiers may be used on explicitly under-aligned memory.

The runtime tracks `__weak` objects which holds non-null values. It is undefined behavior to direct modify a `__weak` object which is being tracked by the runtime except through an objc_storeWeak, objc_destroyWeak, or objc_moveWeak call.

The runtime must provide a number of new entrypoints which the compiler may emit, which are described in the remainder of this section.

Rationale

Several of these functions are semantically equivalent to a message send; we emit calls to C functions instead because:

- the machine code to do so is significantly smaller,
- it is much easier to recognize the C functions in the ARC optimizer, and
- a sufficient sophisticated runtime may be able to avoid the message send in common cases.

Several other of these functions are "fused" operations which can be described entirely in terms of other operations. We use the fused operations primarily as a code-size optimization, although in some cases there is also a real potential for avoiding redundant operations in the runtime.

```
id objc_autorelease(id value);
```

Precondition: `value` is null or a pointer to a valid object.

If `value` is null, this call has no effect. Otherwise, it adds the object to the innermost autorelease pool exactly as if the object had been sent the `autorelease` message.

Always returns `value`.

```
void objc_autoreleasePoolPop(void *pool);
```

Precondition: `pool` is the result of a previous call to objc_autoreleasePoolPush on the current thread, where neither `pool` nor any enclosing pool have previously been popped.

Releases all the objects added to the given autorelease pool and any autorelease pools it encloses, then sets the current autorelease pool to the pool directly enclosing `pool`.

```
void *objc_autoreleasePoolPush(void);
```

Creates a new autorelease pool that is enclosed by the current pool, makes that the current pool, and returns an opaque "handle" to it.

Rationale

While the interface is described as an explicit hierarchy of pools, the rules allow the implementation to just keep a stack of objects, using the stack depth as the opaque pool handle.

```
id objc_autoreleaseReturnValue(id value);
```

Precondition: `value` is null or a pointer to a valid object.

If `value` is null, this call has no effect. Otherwise, it makes a best effort to hand off ownership of a retain count on the object to a call to objc_retainAutoreleasedReturnValue (or objc_unsafeClaimAutoreleasedReturnValue) for the same object in an enclosing call frame. If this is not possible, the object is autoreleased as above.

Always returns `value`.

```
void objc_copyWeak(id *dest, id *src);
```

Precondition: `src` is a valid pointer which either contains a null pointer or has been registered as a `__weak` object. `dest` is a valid pointer which has not been registered as a `__weak` object.

`dest` is initialized to be equivalent to `src`, potentially registering it with the runtime. Equivalent to the following code:

```
void objc_copyWeak(id *dest, id *src) {
  objc_release(objc_initWeak(dest, objc_loadWeakRetained(src)));
}
```

Must be atomic with respect to calls to `objc_storeWeak` on `src`.

```
void objc_destroyWeak(id *object);
```

Precondition: `object` is a valid pointer which either contains a null pointer or has been registered as a `__weak` object.

`object` is unregistered as a weak object, if it ever was. The current value of `object` is left unspecified; otherwise, equivalent to the following code:

```
void objc_destroyWeak(id *object) {
  objc_storeWeak(object, nil);
}
```

Does not need to be atomic with respect to calls to `objc_storeWeak` on `object`.

```
id objc_initWeak(id *object, id value);
```

Precondition: `object` is a valid pointer which has not been registered as a `__weak` object. `value` is null or a pointer to a valid object.

If `value` is a null pointer or the object to which it points has begun deallocation, `object` is zero-initialized. Otherwise, `object` is registered as a `__weak` object pointing to `value`. Equivalent to the following code:

```
id objc_initWeak(id *object, id value) {
  *object = nil;
  return objc_storeWeak(object, value);
}
```

Returns the value of `object` after the call.

Does not need to be atomic with respect to calls to `objc_storeWeak` on `object`.

```
id objc_loadWeak(id *object);
```

Precondition: `object` is a valid pointer which either contains a null pointer or has been registered as a `__weak` object.

If `object` is registered as a `__weak` object, and the last value stored into `object` has not yet been deallocated or begun deallocation, retains and autoreleases that value and returns it. Otherwise returns null. Equivalent to the following code:

```
id objc_loadWeak(id *object) {
  return objc_autorelease(objc_loadWeakRetained(object));
}
```

Must be atomic with respect to calls to `objc_storeWeak` on `object`.

Rationale

Loading weak references would be inherently prone to race conditions without the retain.

```
id objc_loadWeakRetained(id *object);
```

Precondition: `object` is a valid pointer which either contains a null pointer or has been registered as a `__weak` object.

If `object` is registered as a `__weak` object, and the last value stored into `object` has not yet been deallocated or begun deallocation, retains that value and returns it. Otherwise returns null.

Must be atomic with respect to calls to `objc_storeWeak` on `object`.

```
void objc_moveWeak(id *dest, id *src);
```

Precondition: `src` is a valid pointer which either contains a null pointer or has been registered as a `__weak` object. `dest` is a valid pointer which has not been registered as a `__weak` object.

`dest` is initialized to be equivalent to `src`, potentially registering it with the runtime. `src` may then be left in its original state, in which case this call is equivalent to objc_copyWeak, or it may be left as null.

Must be atomic with respect to calls to `objc_storeWeak` on `src`.

```
void objc_release(id value);
```

Precondition: `value` is null or a pointer to a valid object.

If `value` is null, this call has no effect. Otherwise, it performs a release operation exactly as if the object had been sent the `release` message.

```
id objc_retain(id value);
```

Precondition: `value` is null or a pointer to a valid object.

If `value` is null, this call has no effect. Otherwise, it performs a retain operation exactly as if the object had been sent the `retain` message.

Always returns `value`.

```
id objc_retainAutorelease(id value);
```

Precondition: `value` is null or a pointer to a valid object.

If `value` is null, this call has no effect. Otherwise, it performs a retain operation followed by an autorelease operation. Equivalent to the following code:

```
id objc_retainAutorelease(id value) {
  return objc_autorelease(objc_retain(value));
}
```

Always returns `value`.

```
id objc_retainAutoreleaseReturnValue(id value);
```

Precondition: `value` is null or a pointer to a valid object.

If `value` is null, this call has no effect. Otherwise, it performs a retain operation followed by the operation described in objc_autoreleaseReturnValue. Equivalent to the following code:

```
id objc_retainAutoreleaseReturnValue(id value) {
  return objc_autoreleaseReturnValue(objc_retain(value));
}
```

Always returns `value`.

```
id objc_retainAutoreleasedReturnValue(id value);
```

Precondition: `value` is null or a pointer to a valid object.

If `value` is null, this call has no effect. Otherwise, it attempts to accept a hand off of a retain count from a call to objc_autoreleaseReturnValue on `value` in a recently-called function or something it tail-calls. If that fails, it performs a retain operation exactly like objc_retain.

Always returns `value`.

```
id objc_retainBlock(id value);
```

Precondition: `value` is null or a pointer to a valid block object.

If `value` is null, this call has no effect. Otherwise, if the block pointed to by `value` is still on the stack, it is copied to the heap and the address of the copy is returned. Otherwise a retain operation is performed on the block exactly as if it had been sent the `retain` message.

```
void objc_storeStrong(id *object, id value);
```

Precondition: `object` is a valid pointer to a `__strong` object which is adequately aligned for a pointer. `value` is null or a pointer to a valid object.

Performs the complete sequence for assigning to a `__strong` object of non-block type [1]. Equivalent to the following code:

```
void objc_storeStrong(id *object, id value) {
  id oldValue = *object;
  value = [value retain];
  *object = value;
  [oldValue release];
}
```

1 This does not imply that a `__strong` object of block type is an invalid argument to this function. Rather it implies that an `objc_retain` and not an `objc_retainBlock` operation will be emitted if the argument is a block.

```
id objc_storeWeak(id *object, id value);
```

Precondition: `object` is a valid pointer which either contains a null pointer or has been registered as a `__weak` object. `value` is null or a pointer to a valid object.

If `value` is a null pointer or the object to which it points has begun deallocation, `object` is assigned null and unregistered as a `__weak` object. Otherwise, `object` is registered as a `__weak` object or has its registration updated to point to `value`.

Returns the value of `object` after the call.

```
id objc_unsafeClaimAutoreleasedReturnValue(id value);
```

Precondition: `value` is null or a pointer to a valid object.

If `value` is null, this call has no effect. Otherwise, it attempts to accept a hand off of a retain count from a call to objc_autoreleaseReturnValue on `value` in a recently-called function or something it tail-calls (in a manner similar to objc_retainAutoreleasedReturnValue). If that succeeds, it performs a release operation exactly like objc_release. If the handoff fails, this call has no effect.

Always returns `value`.

Matrix Types

Draft Specification 121
- Matrix Type 121
- Matrix Type Attribute 121
- Standard Conversions 121
- Arithmetic Conversions 122
- Matrix Type Element Access Operator 122
- Matrix Type Binary Operators 122
- Matrix Type Builtin Operations 123
- TODOs 124

Decisions for the Implementation in Clang 124

Clang provides a C/C++ language extension that allows users to directly express fixed-size 2-dimensional matrices as language values and perform arithmetic on them.

This feature is currently experimental, and both its design and its implementation are in flux.

Draft Specification

Matrix Type

A matrix type is a scalar type with an underlying *element type*, a constant number of *rows*, and a constant number of *columns*. Matrix types with the same element type, rows, and columns are the same type. A value of a matrix type includes storage for `rows * columns` values of the *element type*. The internal layout, overall size and alignment are implementation-defined.

The maximum of the product of the number of rows and columns is implementation-defined. If that implementation-defined limit is exceeded, the program is ill-formed.

Currently, the element type of a matrix is only permitted to be one of the following types:

- an integer type (as in C2x 6.2.5p19), but excluding enumerated types and `_Bool`
- the standard floating types `float` or `double`
- a half-precision floating point type, if one is supported on the target

Other types may be supported in the future.

Matrix Type Attribute

Matrix types can be declared by adding the `matrix_type` attribute to the declaration of a *typedef* (or a C++ alias declaration). The underlying type of the *typedef* must be a valid matrix element type. The attribute takes two arguments, both of which must be integer constant expressions that evaluate to a value greater than zero. The first specifies the number of rows, and the second specifies the number of columns. The underlying type of the *typedef* becomes a matrix type with the given dimensions and an element type of the former underlying type.

If a declaration of a *typedef-name* has a `matrix_type` attribute, then all declaration of that *typedef-name* shall have a matrix_type attribute with the same element type, number of rows, and number of columns.

Standard Conversions

The standard conversions are extended as follows. Note that these conversions are intentionally not listed as satisfying the constraints for assignment, which is to say, they are only permitted as explicit casts, not as implicit conversions.

A value of matrix type can be converted to another matrix type if the number of rows and columns are the same and the value's elements can be converted to the element type of the result type. The result is a matrix where each element is the converted corresponding element.

A value of any real type (as in C2x 6.2.5p17) can be converted to a matrix type if it can be converted to the element type of the matrix. The result is a matrix where all elements are the converted original value.

If the number of rows or columns differ between the original and resulting type, the program is ill-formed.

Arithmetic Conversions

The usual arithmetic conversions are extended as follows.

Insert at the start:

- If both operands are of matrix type, no arithmetic conversion is performed.
- If one operand is of matrix type and the other operand is of a real type, convert the real type operand to the matrix type according to the standard conversion rules.

Matrix Type Element Access Operator

An expression of the form `E1 [E2] [E3]`, where `E1` has matrix type `cv M`, is a matrix element access expression. Let `T` be the element type of `M`, and let `R` and `C` be the number of rows and columns in `M` respectively. The index expressions shall have integral or unscoped enumeration type and shall not be uses of the comma operator unless parenthesized. The first index expression shall evaluate to a non-negative value less than `R`, and the second index expression shall evaluate to a non-negative value less than `C`, or else the expression has undefined behavior. If `E1` is a prvalue, the result is a prvalue with type `T` and is the value of the element at the given row and column in the matrix. Otherwise, the result is a glvalue with type `cv T` and with the same value category as `E1` which refers to the element at the given row and column in the matrix.

Programs containing a single subscript expression into a matrix are ill-formed.

Note: We considered providing an expression of the form `postfix-expression [expression]` to access columns of a matrix. We think that such an expression would be problematic once both column and row major matrixes are supported: depending on the memory layout, either accessing columns or rows can be done efficiently, but not both. Instead, we propose to provide builtins to extract rows and columns from a matrix. This makes the operations more explicit.

Matrix Type Binary Operators

Given two matrixes, the `+` and `-` operators perform element-wise addition and subtraction, while the `*` operator performs matrix multiplication. `+`, `-`, `*`, and `/` can also be used with a matrix and a scalar value, applying the operation to each element of the matrix.

Earlier versions of this extension did not support division by a scalar. You can test for the availability of this feature with `__has_extension(matrix_types_scalar_division)`.

For the expression `M1 BIN_OP M2` where

- `BIN_OP` is one of `+` or `-`, one of `M1` and `M2` is of matrix type, and the other is of matrix type or real type; or
- `BIN_OP` **is** `*`**, one of** `M1` **and** `M2` **is of matrix type, and the** other is of a real type; or
- `BIN_OP` is `/`, `M1` is of matrix type, and `M2` is of a real type:
- The usual arithmetic conversions are applied to `M1` and `M2`. [Note: if `M1` or `M2` are of a real type, they are broadcast to matrices here. — end note]
- `M1` and `M2` shall be of the same matrix type.
- The result is equivalent to Res in the following where col is the number of columns and row is the number of rows in the matrix type:

```
decltype(M1) Res;
for (int C = 0; C < col; ++C)
  for (int R = 0; R < row; ++R)
    Res[R][C] = M1[R][C] BIN_OP M2[R][C];
```

Given the expression `M1 * M2` where `M1` and `M2` are of matrix type:

- The usual arithmetic conversions are applied to `M1` and `M2`.

- The type of `M1` shall have the same number of columns as the type of `M2` has rows. The element types of `M1` and `M2` shall be the same type.
- The resulting type, `MTy`, is a matrix type with the common element type, the number of rows of `M1` and the number of columns of `M2`.
- The result is equivalent to `Res` in the following where `EltTy` is the element type of `MTy`, `col` is the number of columns, `row` is the number of rows in `MTy` and `inner` is the number of columns of `M1`:

```
MTy Res;
for (int C = 0; C < col; ++C) {
  for (int R = 0; R < row; ++R) {
    EltTy Elt = 0;
    for (int K = 0; K < inner; ++K) {
      Elt += M1[R][K] * M2[K][C];
  }
  Res[R][C] = Elt;
}
```

All operations on matrix types match the behavior of the element type with respect to signed overflows.

With respect to floating-point contraction, rounding and environment rules, operations on matrix types match the behavior of the elementwise operations in the corresponding expansions provided above.

Operations on floating-point matrices have the same rounding and floating-point environment behavior as ordinary floating-point operations in the expression's context. For the purposes of floating-point contraction, all calculations done as part of a matrix operation are considered intermediate operations, and their results need not be rounded to the format of the element type until the final result in the containing expression. This is subject to the normal restrictions on contraction, such as `#pragma STDC FP_CONTRACT`.

For the +=, -= and *= operators the semantics match their expanded variants.

Matrix Type Builtin Operations

Each matrix type supports a collection of builtin expressions that look like function calls but do not form an overload set. Here they are described as function declarations with rules for how to construct the argument list types and return type and the library description elements from [library.description.structure.specifications]/3 in the C++ standard.

Definitions:

- *M*, *M1*, *M2*, *M3* - Matrix types
- *T* - Element type
- *row*, *col* - Row and column arguments respectively.

```
M2 __builtin_matrix_transpose(M1 matrix)
```

Remarks: The return type is a cv-unqualified matrix type that has the same element type as `M1` and has the same number of rows as `M1` has columns and the same number of columns as `M1` has rows.

Returns: A matrix `Res` equivalent to the code below, where `col` refers to the number of columns of `M`, and `row` to the number of rows of `M`.

Effects: Equivalent to:

```
M Res;
for (int C = 0; C < col; ++C)
  for (int R = 0; R < row; ++R)
    Res[C][R] = matrix[R][C];
```

```
M __builtin_matrix_column_major_load(T *ptr, size_t row, size_t col, size_t columnStr
ide)
```

Mandates: `row` and `col` shall be integral constants greater than 0.

Preconditions: `columnStride` is greater than or equal to `row`.

Remarks: The return type is a cv-unqualified matrix type with an element type of the cv-unqualified version of `T` and a number of rows and columns equal to `row` and `col` respectively. The parameter `columnStride` is optional and if omitted `row` is used as `columnStride`.

Returns: A matrix `Res` equivalent to:

```
M Res;
for (size_t C = 0; C < col; ++C) {
  for (size_t R = 0; R < row; ++K)
    Res[R][C] = ptr[R];
  ptr += columnStride
}
```

`void __builtin_matrix_column_major_store(M matrix, T *ptr, size_t columnStride)`

Preconditions: `columnStride` is greater than or equal to the number of rows in `M`.

Remarks: The type `T` is the const-unqualified version of the matrix argument's element type. The parameter `columnStride` is optional and if omitted, the number of rows of `M` is used as `columnStride`.

Effects: Equivalent to:

```
for (size_t C = 0; C < columns in M; ++C) {
  for (size_t R = 0; R < rows in M; ++K)
    ptr[R] = matrix[R][C];
  ptr += columnStride
}
```

TODOs

TODO: Does it make sense to allow M::element_type, M::rows, and M::columns where M is a matrix type? We don't support this anywhere else, but it's convenient. The alternative is using template deduction to extract this information. Also add spelling for C.

Future Work: Initialization syntax.

Decisions for the Implementation in Clang

This section details decisions taken for the implementation in Clang and is not part of the draft specification.

The elements of a value of a matrix type are laid out in column-major order without padding.

We propose to provide a Clang option to override this behavior and allow contraction of those operations (e.g. *-ffp-contract=matrix*).

TODO: Specify how matrix values are passed to functions.

Introduction

This document describes the language extensions provided by Clang. In addition to the language extensions listed here, Clang aims to support a broad range of GCC extensions. Please see the GCC manual for more information on these extensions.

Feature Checking Macros

Language extensions can be very useful, but only if you know you can depend on them. In order to allow fine-grain features checks, we support three builtin function-like macros. This allows you to directly test for a feature in your code without having to resort to something like autoconf or fragile "compiler version checks".

`__has_builtin`

This function-like macro takes a single identifier argument that is the name of a builtin function, a builtin pseudo-function (taking one or more type arguments), or a builtin template. It evaluates to 1 if the builtin is supported or 0 if not. It can be used like this:

```
#ifndef __has_builtin         // Optional of course.
  #define __has_builtin(x) 0  // Compatibility with non-clang compilers.
#endif

...
#if __has_builtin(__builtin_trap)
  __builtin_trap();
#else
  abort();
#endif
...
```

Note

Prior to Clang 10, `__has_builtin` could not be used to detect most builtin pseudo-functions.

`__has_builtin` should not be used to detect support for a builtin macro; use `#ifdef` instead.

`__has_feature` and `__has_extension`

These function-like macros take a single identifier argument that is the name of a feature. `__has_feature` evaluates to 1 if the feature is both supported by Clang and standardized in the current language standard or 0 if not (but see below), while `__has_extension` evaluates to 1 if the feature is supported by Clang in the current language (either as a language extension or a standard language feature) or 0 if not. They can be used like this:

```
#ifndef __has_feature         // Optional of course.
  #define __has_feature(x) 0  // Compatibility with non-clang compilers.
#endif
#ifndef __has_extension
  #define __has_extension __has_feature // Compatibility with pre-3.0 compilers.
#endif

...
#if __has_feature(cxx_rvalue_references)
// This code will only be compiled with the -std=c++11 and -std=gnu++11
// options, because rvalue references are only standardized in C++11.
#endif

#if __has_extension(cxx_rvalue_references)
// This code will be compiled with the -std=c++11, -std=gnu++11, -std=c++98
// and -std=gnu++98 options, because rvalue references are supported as a
// language extension in C++98.
#endif
```

For backward compatibility, `__has_feature` can also be used to test for support for non-standardized features, i.e. features not prefixed `c_`, `cxx_` or `objc_`.

Another use of `__has_feature` is to check for compiler features not related to the language standard, such as e.g. AddressSanitizer.

If the `-pedantic-errors` option is given, `__has_extension` is equivalent to `__has_feature`.

The feature tag is described along with the language feature below.

The feature name or extension name can also be specified with a preceding and following __ (double underscore) to avoid interference from a macro with the same name. For instance, `__cxx_rvalue_references__` can be used instead of `cxx_rvalue_references`.

__has_cpp_attribute

This function-like macro is available in C++20 by default, and is provided as an extension in earlier language standards. It takes a single argument that is the name of a double-square-bracket-style attribute. The argument can either be a single identifier or a scoped identifier. If the attribute is supported, a nonzero value is returned. If the attribute is a standards-based attribute, this macro returns a nonzero value based on the year and month in which the attribute was voted into the working draft. See WG21 SD-6 for the list of values returned for standards-based attributes. If the attribute is not supported by the current compilation target, this macro evaluates to 0. It can be used like this:

```
#ifndef __has_cpp_attribute         // For backwards compatibility
  #define __has_cpp_attribute(x) 0
#endif

...
#if __has_cpp_attribute(clang::fallthrough)
#define FALLTHROUGH [[clang::fallthrough]]
#else
#define FALLTHROUGH
#endif
...
```

The attribute scope tokens `clang` and `_Clang` are interchangeable, as are the attribute scope tokens `gnu` and `__gnu__`. Attribute tokens in either of these namespaces can be specified with a preceding and following `__` (double underscore) to avoid interference from a macro with the same name. For instance, `gnu::__const__` can be used instead of `gnu::const`.

__has_c_attribute

This function-like macro takes a single argument that is the name of an attribute exposed with the double square-bracket syntax in C mode. The argument can either be a single identifier or a scoped identifier. If the attribute is supported, a nonzero value is returned. If the attribute is not supported by the current compilation target, this macro evaluates to 0. It can be used like this:

```
#ifndef __has_c_attribute         // Optional of course.
  #define __has_c_attribute(x) 0  // Compatibility with non-clang compilers.
#endif

...
#if __has_c_attribute(fallthrough)
  #define FALLTHROUGH [[fallthrough]]
#else
  #define FALLTHROUGH
#endif
...
```

The attribute scope tokens `clang` and `_Clang` are interchangeable, as are the attribute scope tokens `gnu` and `__gnu__`. Attribute tokens in either of these namespaces can be specified with a preceding and following `__` (double underscore) to avoid interference from a macro with the same name. For instance, `gnu::__const__` can be used instead of `gnu::const`.

__has_attribute

This function-like macro takes a single identifier argument that is the name of a GNU-style attribute. It evaluates to 1 if the attribute is supported by the current compilation target, or 0 if not. It can be used like this:

```
#ifndef __has_attribute         // Optional of course.
  #define __has_attribute(x) 0  // Compatibility with non-clang compilers.
#endif

...
#if __has_attribute(always_inline)
#define ALWAYS_INLINE __attribute__((always_inline))
```

```
#else
#define ALWAYS_INLINE
#endif
...
```

The attribute name can also be specified with a preceding and following __ (double underscore) to avoid interference from a macro with the same name. For instance, `__always_inline__` can be used instead of `always_inline`.

`__has_declspec_attribute`

This function-like macro takes a single identifier argument that is the name of an attribute implemented as a Microsoft-style `__declspec` attribute. It evaluates to 1 if the attribute is supported by the current compilation target, or 0 if not. It can be used like this:

```
#ifndef __has_declspec_attribute         // Optional of course.
  #define __has_declspec_attribute(x) 0  // Compatibility with non-clang compilers.
#endif

...
#if __has_declspec_attribute(dllexport)
#define DLLEXPORT __declspec(dllexport)
#else
#define DLLEXPORT
#endif
...
```

The attribute name can also be specified with a preceding and following __ (double underscore) to avoid interference from a macro with the same name. For instance, `__dllexport__` can be used instead of `dllexport`.

`__is_identifier`

This function-like macro takes a single identifier argument that might be either a reserved word or a regular identifier. It evaluates to 1 if the argument is just a regular identifier and not a reserved word, in the sense that it can then be used as the name of a user-defined function or variable. Otherwise it evaluates to 0. It can be used like this:

```
...
#ifdef __is_identifier          // Compatibility with non-clang compilers.
  #if __is_identifier(__wchar_t)
    typedef wchar_t __wchar_t;
  #endif
#endif

__wchar_t WideCharacter;
...
```

Include File Checking Macros

Not all developments systems have the same include files. The __has_include and __has_include_next macros allow you to check for the existence of an include file before doing a possibly failing `#include` directive. Include file checking macros must be used as expressions in `#if` or `#elif` preprocessing directives.

`__has_include`

This function-like macro takes a single file name string argument that is the name of an include file. It evaluates to 1 if the file can be found using the include paths, or 0 otherwise:

```
// Note the two possible file name string formats.
#if __has_include("myinclude.h") && __has_include(<stdint.h>)
# include "myinclude.h"
#endif
```

To test for this feature, use `#if defined(__has_include)`:

```
// To avoid problem with non-clang compilers not having this macro.
#if defined(__has_include)
#if __has_include("myinclude.h")
# include "myinclude.h"
#endif
#endif
```

`__has_include_next`

This function-like macro takes a single file name string argument that is the name of an include file. It is like `__has_include` except that it looks for the second instance of the given file found in the include paths. It evaluates to 1 if the second instance of the file can be found using the include paths, or 0 otherwise:

```
// Note the two possible file name string formats.
#if __has_include_next("myinclude.h") && __has_include_next(<stdint.h>)
# include_next "myinclude.h"
#endif

// To avoid problem with non-clang compilers not having this macro.
#if defined(__has_include_next)
#if __has_include_next("myinclude.h")
# include_next "myinclude.h"
#endif
#endif
```

Note that `__has_include_next`, like the GNU extension `#include_next` directive, is intended for use in headers only, and will issue a warning if used in the top-level compilation file. A warning will also be issued if an absolute path is used in the file argument.

`__has_warning`

This function-like macro takes a string literal that represents a command line option for a warning and returns true if that is a valid warning option.

```
#if __has_warning("-Wformat")
...
#endif
```

Builtin Macros

`__BASE_FILE__`

Defined to a string that contains the name of the main input file passed to Clang.

`__FILE_NAME__`

Clang-specific extension that functions similar to `__FILE__` but only renders the last path component (the filename) instead of an invocation dependent full path to that file.

`__COUNTER__`

Defined to an integer value that starts at zero and is incremented each time the `__COUNTER__` macro is expanded.

`__INCLUDE_LEVEL__`

Defined to an integral value that is the include depth of the file currently being translated. For the main file, this value is zero.

`__TIMESTAMP__`

Defined to the date and time of the last modification of the current source file.

`__clang__`

Defined when compiling with Clang

`__clang_major__`

Defined to the major marketing version number of Clang (e.g., the 2 in 2.0.1). Note that marketing version numbers should not be used to check for language features, as different vendors use different numbering schemes. Instead, use the Feature Checking Macros.

`__clang_minor__`

Defined to the minor version number of Clang (e.g., the 0 in 2.0.1). Note that marketing version numbers should not be used to check for language features, as different vendors use different numbering schemes. Instead, use the Feature Checking Macros.

`__clang_patchlevel__`

Defined to the marketing patch level of Clang (e.g., the 1 in 2.0.1).

`__clang_version__`

Defined to a string that captures the Clang marketing version, including the Subversion tag or revision number, e.g., "`1.5 (trunk 102332)`".

`__clang_literal_encoding__`

Defined to a narrow string literal that represents the current encoding of narrow string literals, e.g., `"hello"`. This macro typically expands to "UTF-8" (but may change in the future if the `-fexec-charset="Encoding-Name"` option is implemented.)

`__clang_wide_literal_encoding__`

Defined to a narrow string literal that represents the current encoding of wide string literals, e.g., `L"hello"`. This macro typically expands to "UTF-16" or "UTF-32" (but may change in the future if the `-fwide-exec-charset="Encoding-Name"` option is implemented.)

Vectors and Extended Vectors

Supports the GCC, OpenCL, AltiVec and NEON vector extensions.

OpenCL vector types are created using the `ext_vector_type` attribute. It supports the `V.xyzw` syntax and other tidbits as seen in OpenCL. An example is:

```
typedef float float4 __attribute__((ext_vector_type(4)));
typedef float float2 __attribute__((ext_vector_type(2)));

float4 foo(float2 a, float2 b) {
  float4 c;
  c.xz = a;
  c.yw = b;
  return c;
}
```

Query for this feature with `__has_attribute(ext_vector_type)`.

Giving `-maltivec` option to clang enables support for AltiVec vector syntax and functions. For example:

```
vector float foo(vector int a) {
  vector int b;
  b = vec_add(a, a) + a;
  return (vector float)b;
}
```

NEON vector types are created using `neon_vector_type` and `neon_polyvector_type` attributes. For example:

```
typedef __attribute__((neon_vector_type(8))) int8_t int8x8_t;
typedef __attribute__((neon_polyvector_type(16))) poly8_t poly8x16_t;

int8x8_t foo(int8x8_t a) {
  int8x8_t v;
  v = a;
  return v;
}
```

GCC vector types are created using the `vector_size(N)` attribute. The argument `N` specifies the number of bytes that will be allocated for an object of this type. The size has to be multiple of the size of the vector element type. For example:

```
// OK: This declares a vector type with four 'int' elements
typedef int int4 __attribute__((vector_size(4 * sizeof(int))));

// ERROR: '11' is not a multiple of sizeof(int)
typedef int int_impossible __attribute__((vector_size(11)));

int4 foo(int4 a) {
  int4 v;
  v = a;
  return v;
}
```

Boolean Vectors

Clang also supports the ext_vector_type attribute with boolean element types in C and C++. For example:

```
// legal for Clang, error for GCC:
typedef bool bool4 __attribute__((ext_vector_type(4)));
// Objects of bool4 type hold 8 bits, sizeof(bool4) == 1

bool4 foo(bool4 a) {
  bool4 v;
  v = a;
  return v;
}
```

Boolean vectors are a Clang extension of the ext vector type. Boolean vectors are intended, though not guaranteed, to map to vector mask registers. The size parameter of a boolean vector type is the number of bits in the vector. The boolean vector is dense and each bit in the boolean vector is one vector element.

The semantics of boolean vectors borrows from C bit-fields with the following differences:

- Distinct boolean vectors are always distinct memory objects (there is no packing).
- Only the operators *?:*, *!*, ~, |, *&*, ^ and comparison are allowed on boolean vectors.
- Casting a scalar bool value to a boolean vector type means broadcasting the scalar value onto all lanes (same as general ext_vector_type).
- It is not possible to access or swizzle elements of a boolean vector (different than general ext_vector_type).

The size and alignment are both the number of bits rounded up to the next power of two, but the alignment is at most the maximum vector alignment of the target.

Vector Literals

Vector literals can be used to create vectors from a set of scalars, or vectors. Either parentheses or braces form can be used. In the parentheses form the number of literal values specified must be one, i.e. referring to a scalar value, or must match the size of the vector type being created. If a single scalar literal value is specified, the scalar literal value will be replicated to all the components of the vector type. In the brackets form any number of literals can be specified. For example:

```
typedef int v4si __attribute__((__vector_size__(16)));
typedef float float4 __attribute__((ext_vector_type(4)));
typedef float float2 __attribute__((ext_vector_type(2)));

v4si vsi = (v4si){1, 2, 3, 4};
float4 vf = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
vector int vi1 = (vector int)(1);    // vi1 will be (1, 1, 1, 1).
vector int vi2 = (vector int){1};    // vi2 will be (1, 0, 0, 0).
vector int vi3 = (vector int)(1, 2); // error
```

```
vector int vi4 = (vector int){1, 2}; // vi4 will be (1, 2, 0, 0).
vector int vi5 = (vector int)(1, 2, 3, 4);
float4 vf = (float4)((float2)(1.0f, 2.0f), (float2)(3.0f, 4.0f));
```

Vector Operations

The table below shows the support for each operation by vector extension. A dash indicates that an operation is not accepted according to a corresponding specification.

Operator	OpenCL	AltiVec	GCC	NEON
[]	yes	yes	yes	–
unary operators +, –	yes	yes	yes	–
++, – –	yes	yes	yes	–
+,–,*,/,%	yes	yes	yes	–
bitwise operators &,|,^,~	yes	yes	yes	–
>>,<<	yes	yes	yes	–
!, &&, ||	yes	–	yes	–
==, !=, >, <, >=, <=	yes	yes	yes	–
=	yes	yes	yes	yes
?: [2]	yes	–	yes	–
sizeof	yes	yes	yes	yes
C-style cast	yes	yes	yes	no
reinterpret_cast	yes	no	yes	no
static_cast	yes	no	yes	no
const_cast	no	no	no	no
address &v[i]	no	no	no [3]	no

See also __builtin_shufflevector, __builtin_convertvector.

2 ternary operator(?:) has different behaviors depending on condition operand's vector type. If the condition is a GNU vector (i.e. __vector_size__), it's only available in C++ and uses normal bool conversions (that is, != 0). If it's an extension (OpenCL) vector, it's only available in C and OpenCL C. And it selects base on signedness of the condition operands (OpenCL v1.1 s6.3.9).

3 Clang does not allow the address of an element to be taken while GCC allows this. This is intentional for vectors with a boolean element type and not implemented otherwise.

Vector Builtins

Note: The implementation of vector builtins is work-in-progress and incomplete.

In addition to the operators mentioned above, Clang provides a set of builtins to perform additional operations on certain scalar and vector types.

Let T be one of the following types:

- an integer type (as in C2x 6.2.5p19), but excluding enumerated types and _Bool
- the standard floating types float or double
- a half-precision floating point type, if one is supported on the target
- a vector type.

For scalar types, consider the operation applied to a vector with a single element.

Elementwise Builtins

Each builtin returns a vector equivalent to applying the specified operation elementwise to the input.

Unless specified otherwise operation(±0) = ±0 and operation(±infinity) = ±infinity

Name	Operation	Supported element types
T __builtin_elementwise_abs(T x)	return the absolute value of a number x; the absolute value of the most negative integer remains the most negative integer	signed integer and floating point types
T __builtin_elementwise_ceil(T x)	return the smallest integral value greater than or equal to x	floating point types
T __builtin_elementwise_floor(T x)	return the largest integral value less than or equal to x	floating point types
T __builtin_elementwise_roundeven(T x)	round x to the nearest integer value in floating point format, rounding halfway cases to even (that is, to the nearest value that is an even integer), regardless of the current rounding direction.	floating point types
T__builtin_elementwise_trunc(T x)	return the integral value nearest to but no larger in magnitude than x	floating point types
T __builtin_elementwise_max(T x, T y)	return x or y, whichever is larger	integer and floating point types
T __builtin_elementwise_min(T x, T y)	return x or y, whichever is smaller	integer and floating point types
T __builtin_elementwise_add_sat(T x, T y)	return the sum of x and y, clamped to the range of representable values for the signed/unsigned integer type.	integer types
T __builtin_elementwise_sub_sat(T x, T y)	return the difference of x and y, clamped to the range of representable values for the signed/unsigned integer type.	integer types

Reduction Builtins

Each builtin returns a scalar equivalent to applying the specified operation(x, y) as recursive even-odd pairwise reduction to all vector elements. `operation(x, y)` is repeatedly applied to each non-overlapping even-odd element pair with indices `i * 2` and `i * 2 + 1` with `i in [0, Number of elements / 2)`. If the numbers of elements is not a power of 2, the vector is widened with neutral elements for the reduction at the end to the next power of 2.

Example:

```
__builtin_reduce_add([e3, e2, e1, e0]) = __builtin_reduced_add([e3 + e2, e1 + e0])
                                       = (e3 + e2) + (e1 + e0)
```

Let `VT` be a vector type and `ET` the element type of `VT`.

Name	Operation	Supported element types
ET __builtin_reduce_max(VT a)	return x or y, whichever is larger; If exactly one argument is a NaN, return the other argument. If both arguments are NaNs, fmax() return a NaN.	integer and floating point types
ET __builtin_reduce_min(VT a)	return x or y, whichever is smaller; If exactly one argument is a NaN, return the other argument. If both arguments are NaNs, fmax() return a NaN.	integer and floating point types
ET __builtin_reduce_add(VT a)	+	integer and floating point types

Name	Operation	Supported element types
ET __builtin_reduce_mul(VT a)	*	integer and floating point types
ET __builtin_reduce_and(VT a)	&	integer types
ET __builtin_reduce_or(VT a)	|	integer types
ET __builtin_reduce_xor(VT a)	^	integer types

Matrix Types

Clang provides an extension for matrix types, which is currently being implemented. See the draft specification for more details.

For example, the code below uses the matrix types extension to multiply two 4x4 float matrices and add the result to a third 4x4 matrix.

```
typedef float m4x4_t __attribute__((matrix_type(4, 4)));

m4x4_t f(m4x4_t a, m4x4_t b, m4x4_t c) {
  return a + b * c;
}
```

The matrix type extension also supports operations on a matrix and a scalar.

```
typedef float m4x4_t __attribute__((matrix_type(4, 4)));

m4x4_t f(m4x4_t a) {
  return (a + 23) * 12;
}
```

The matrix type extension supports division on a matrix and a scalar but not on a matrix and a matrix.

```
typedef float m4x4_t __attribute__((matrix_type(4, 4)));

m4x4_t f(m4x4_t a) {
  a = a / 3.0;
  return a;
}
```

The matrix type extension supports compound assignments for addition, subtraction, and multiplication on matrices and on a matrix and a scalar, provided their types are consistent.

```
typedef float m4x4_t __attribute__((matrix_type(4, 4)));

m4x4_t f(m4x4_t a, m4x4_t b) {
  a += b;
  a -= b;
  a *= b;
  a += 23;
  a -= 12;
  return a;
}
```

The matrix type extension supports explicit casts. Implicit type conversion between matrix types is not allowed.

```
typedef int ix5x5 __attribute__((matrix_type(5, 5)));
typedef float fx5x5 __attribute__((matrix_type(5, 5)));

fx5x5 f1(ix5x5 i, fx5x5 f) {
  return (fx5x5) i;
```

```
}


template <typename X>
using matrix_4_4 = X __attribute__((matrix_type(4, 4)));

void f2() {
  matrix_5_5<double> d;
  matrix_5_5<int> i;
  i = (matrix_5_5<int>)d;
  i = static_cast<matrix_5_5<int>>(d);
}
```

Half-Precision Floating Point

Clang supports three half-precision (16-bit) floating point types: `__fp16`, `_Float16` and `__bf16`. These types are supported in all language modes.

`__fp16` is supported on every target, as it is purely a storage format; see below. `_Float16` is currently only supported on the following targets, with further targets pending ABI standardization:

- 32-bit ARM
- 64-bit ARM (AArch64)
- AMDGPU
- SPIR
- X86 (Only available under feature AVX512-FP16)

`_Float16` will be supported on more targets as they define ABIs for it.

`__bf16` is purely a storage format; it is currently only supported on the following targets: * 32-bit ARM * 64-bit ARM (AArch64)

The `__bf16` type is only available when supported in hardware.

`__fp16` is a storage and interchange format only. This means that values of `__fp16` are immediately promoted to (at least) `float` when used in arithmetic operations, so that e.g. the result of adding two `__fp16` values has type `float`. The behavior of `__fp16` is specified by the ARM C Language Extensions (ACLE). Clang uses the `binary16` format from IEEE 754-2008 for `__fp16`, not the ARM alternative format.

`_Float16` is an interchange floating-point type. This means that, just like arithmetic on `float` or `double`, arithmetic on `_Float16` operands is formally performed in the `_Float16` type, so that e.g. the result of adding two `_Float16` values has type `_Float16`. The behavior of `_Float16` is specified by ISO/IEC TS 18661-3:2015 (“Floating-point extensions for C”). As with `__fp16`, Clang uses the `binary16` format from IEEE 754-2008 for `_Float16`.

`_Float16` arithmetic will be performed using native half-precision support when available on the target (e.g. on ARMv8.2a); otherwise it will be performed at a higher precision (currently always `float`) and then truncated down to `_Float16`. Note that C and C++ allow intermediate floating-point operands of an expression to be computed with greater precision than is expressible in their type, so Clang may avoid intermediate truncations in certain cases; this may lead to results that are inconsistent with native arithmetic.

It is recommended that portable code use `_Float16` instead of `__fp16`, as it has been defined by the C standards committee and has behavior that is more familiar to most programmers.

Because `__fp16` operands are always immediately promoted to `float`, the common real type of `__fp16` and `_Float16` for the purposes of the usual arithmetic conversions is `float`.

A literal can be given `_Float16` type using the suffix `f16`. For example, `3.14f16`.

Because default argument promotion only applies to the standard floating-point types, `_Float16` values are not promoted to `double` when passed as variadic or untyped arguments. As a consequence, some caution must be taken when using certain library facilities with `_Float16`; for example, there is no `printf` format specifier for `_Float16`, and (unlike `float`) it will not be implicitly promoted to `double` when passed to `printf`, so the programmer must explicitly cast it to `double` before using it with an `%f` or similar specifier.

Messages on `deprecated` and `unavailable` Attributes

An optional string message can be added to the `deprecated` and `unavailable` attributes. For example:

```
void explode(void) __attribute__((deprecated("extremely unsafe, use 'combust' instead!!!")));
```

If the deprecated or unavailable declaration is used, the message will be incorporated into the appropriate diagnostic:

```
harmless.c:4:3: warning: 'explode' is deprecated: extremely unsafe, use 'combust' instead!!!
      [-Wdeprecated-declarations]
  explode();
  ^
```

Query for this feature with `__has_extension(attribute_deprecated_with_message)` and `__has_extension(attribute_unavailable_with_message)`.

Attributes on Enumerators

Clang allows attributes to be written on individual enumerators. This allows enumerators to be deprecated, made unavailable, etc. The attribute must appear after the enumerator name and before any initializer, like so:

```
enum OperationMode {
  OM_Invalid,
  OM_Normal,
  OM_Terrified __attribute__((deprecated)),
  OM_AbortOnError __attribute__((deprecated)) = 4
};
```

Attributes on the `enum` declaration do not apply to individual enumerators.

Query for this feature with `__has_extension(enumerator_attributes)`.

C++11 Attributes on using-declarations

Clang allows C++-style `[[]]` attributes to be written on using-declarations. For instance:

```
[[clang::using_if_exists]] using foo::bar;
using foo::baz [[clang::using_if_exists]];
```

You can test for support for this extension with `__has_extension(cxx_attributes_on_using_declarations)`.

‘User-Specified’ System Frameworks

Clang provides a mechanism by which frameworks can be built in such a way that they will always be treated as being “system frameworks”, even if they are not present in a system framework directory. This can be useful to system framework developers who want to be able to test building other applications with development builds of their framework, including the manner in which the compiler changes warning behavior for system headers.

Framework developers can opt-in to this mechanism by creating a “`.system_framework`” file at the top-level of their framework. That is, the framework should have contents like:

```
.../TestFramework.framework
.../TestFramework.framework/.system_framework
.../TestFramework.framework/Headers
.../TestFramework.framework/Headers/TestFramework.h
...
```

Clang will treat the presence of this file as an indicator that the framework should be treated as a system framework, regardless of how it was found in the framework search path. For consistency, we recommend that such files never be included in installed versions of the framework.

Checks for Standard Language Features

The `__has_feature` macro can be used to query if certain standard language features are enabled. The `__has_extension` macro can be used to query if language features are available as an extension when compiling for a standard which does not provide them. The features which can be tested are listed here.

Since Clang 3.4, the C++ SD-6 feature test macros are also supported. These are macros with names of the form `__cpp_<feature_name>`, and are intended to be a portable way to query the supported features of the compiler. See the C++ status page for information on the version of SD-6 supported by each Clang release, and the macros provided by that revision of the recommendations.

C++98

The features listed below are part of the C++98 standard. These features are enabled by default when compiling C++ code.

C++ exceptions

Use `__has_feature(cxx_exceptions)` to determine if C++ exceptions have been enabled. For example, compiling code with `-fno-exceptions` disables C++ exceptions.

C++ RTTI

Use `__has_feature(cxx_rtti)` to determine if C++ RTTI has been enabled. For example, compiling code with `-fno-rtti` disables the use of RTTI.

C++11

The features listed below are part of the C++11 standard. As a result, all these features are enabled with the `-std=c++11` or `-std=gnu++11` option when compiling C++ code.

C++11 SFINAE includes access control

Use `__has_feature(cxx_access_control_sfinae)` or `__has_extension(cxx_access_control_sfinae)` to determine whether access-control errors (e.g., calling a private constructor) are considered to be template argument deduction errors (aka SFINAE errors), per C++ DR1170.

C++11 alias templates

Use `__has_feature(cxx_alias_templates)` or `__has_extension(cxx_alias_templates)` to determine if support for C++11's alias declarations and alias templates is enabled.

C++11 alignment specifiers

Use `__has_feature(cxx_alignas)` or `__has_extension(cxx_alignas)` to determine if support for alignment specifiers using `alignas` is enabled.

Use `__has_feature(cxx_alignof)` or `__has_extension(cxx_alignof)` to determine if support for the `alignof` keyword is enabled.

C++11 attributes

Use `__has_feature(cxx_attributes)` or `__has_extension(cxx_attributes)` to determine if support for attribute parsing with C++11's square bracket notation is enabled.

C++11 generalized constant expressions

Use `__has_feature(cxx_constexpr)` to determine if support for generalized constant expressions (e.g., `constexpr`) is enabled.

C++11 `decltype()`

Use `__has_feature(cxx_decltype)` or `__has_extension(cxx_decltype)` to determine if support for the `decltype()` specifier is enabled. C++11's `decltype` does not require type-completeness of a function call expression. Use `__has_feature(cxx_decltype_incomplete_return_types)` or `__has_extension(cxx_decltype_incomplete_return_types)` to determine if support for this feature is enabled.

C++11 default template arguments in function templates

Use `__has_feature(cxx_default_function_template_args)` or `__has_extension(cxx_default_function_template_args)` to determine if support for default template arguments in function templates is enabled.

C++11 `defaulted` functions

Use `__has_feature(cxx_defaulted_functions)` or `__has_extension(cxx_defaulted_functions)` to determine if support for defaulted function definitions (with `= default`) is enabled.

C++11 delegating constructors

Use `__has_feature(cxx_delegating_constructors)` to determine if support for delegating constructors is enabled.

C++11 `deleted` functions

Use `__has_feature(cxx_deleted_functions)` or `__has_extension(cxx_deleted_functions)` to determine if support for deleted function definitions (with `= delete`) is enabled.

C++11 explicit conversion functions

Use `__has_feature(cxx_explicit_conversions)` to determine if support for `explicit` conversion functions is enabled.

C++11 generalized initializers

Use `__has_feature(cxx_generalized_initializers)` to determine if support for generalized initializers (using braced lists and `std::initializer_list`) is enabled.

C++11 implicit move constructors/assignment operators

Use `__has_feature(cxx_implicit_moves)` to determine if Clang will implicitly generate move constructors and move assignment operators where needed.

C++11 inheriting constructors

Use `__has_feature(cxx_inheriting_constructors)` to determine if support for inheriting constructors is enabled.

C++11 inline namespaces

Use `__has_feature(cxx_inline_namespaces)` or `__has_extension(cxx_inline_namespaces)` to determine if support for inline namespaces is enabled.

C++11 lambdas

Use `__has_feature(cxx_lambdas)` or `__has_extension(cxx_lambdas)` to determine if support for lambdas is enabled.

C++11 local and unnamed types as template arguments

Use `__has_feature(cxx_local_type_template_args)` or `__has_extension(cxx_local_type_template_args)` to determine if support for local and unnamed types as template arguments is enabled.

C++11 noexcept

Use `__has_feature(cxx_noexcept)` or `__has_extension(cxx_noexcept)` to determine if support for noexcept exception specifications is enabled.

C++11 in-class non-static data member initialization

Use `__has_feature(cxx_nonstatic_member_init)` to determine whether in-class initialization of non-static data members is enabled.

C++11 `nullptr`

Use `__has_feature(cxx_nullptr)` or `__has_extension(cxx_nullptr)` to determine if support for `nullptr` is enabled.

C++11 `override control`

Use `__has_feature(cxx_override_control)` or `__has_extension(cxx_override_control)` to determine if support for the override control keywords is enabled.

C++11 reference-qualified functions

Use `__has_feature(cxx_reference_qualified_functions)` or `__has_extension(cxx_reference_qualified_functions)` to determine if support for reference-qualified functions (e.g., member functions with `&` or `&&` applied to `*this`) is enabled.

C++11 range-based `for` loop

Use `__has_feature(cxx_range_for)` or `__has_extension(cxx_range_for)` to determine if support for the range-based for loop is enabled.

C++11 raw string literals

Use `__has_feature(cxx_raw_string_literals)` to determine if support for raw string literals (e.g., `R"x(foo\bar)x"`) is enabled.

C++11 rvalue references

Use `__has_feature(cxx_rvalue_references)` or `__has_extension(cxx_rvalue_references)` to determine if support for rvalue references is enabled.

C++11 `static_assert()`

Use `__has_feature(cxx_static_assert)` or `__has_extension(cxx_static_assert)` to determine if support for compile-time assertions using `static_assert` is enabled.

C++11 `thread_local`

Use `__has_feature(cxx_thread_local)` to determine if support for `thread_local` variables is enabled.

C++11 type inference

Use `__has_feature(cxx_auto_type)` or `__has_extension(cxx_auto_type)` to determine C++11 type inference is supported using the `auto` specifier. If this is disabled, `auto` will instead be a storage class specifier, as in C or C++98.

C++11 strongly typed enumerations

Use `__has_feature(cxx_strong_enums)` or `__has_extension(cxx_strong_enums)` to determine if support for strongly typed, scoped enumerations is enabled.

C++11 trailing return type

Use `__has_feature(cxx_trailing_return)` or `__has_extension(cxx_trailing_return)` to determine if support for the alternate function declaration syntax with trailing return type is enabled.

C++11 Unicode string literals

Use `__has_feature(cxx_unicode_literals)` to determine if support for Unicode string literals is enabled.

C++11 unrestricted unions

Use `__has_feature(cxx_unrestricted_unions)` to determine if support for unrestricted unions is enabled.

C++11 user-defined literals

Use `__has_feature(cxx_user_literals)` to determine if support for user-defined literals is enabled.

C++11 variadic templates

Use `__has_feature(cxx_variadic_templates)` or `__has_extension(cxx_variadic_templates)` to determine if support for variadic templates is enabled.

C++14

The features listed below are part of the C++14 standard. As a result, all these features are enabled with the `-std=C++14` or `-std=gnu++14` option when compiling C++ code.

C++14 binary literals

Use `__has_feature(cxx_binary_literals)` or `__has_extension(cxx_binary_literals)` to determine whether binary literals (for instance, `0b10010`) are recognized. Clang supports this feature as an extension in all language modes.

C++14 contextual conversions

Use `__has_feature(cxx_contextual_conversions)` or `__has_extension(cxx_contextual_conversions)` to determine if the C++14 rules are used when performing an implicit conversion for an array bound in a *new-expression*, the operand of a *delete-expression*, an integral constant expression, or a condition in a `switch` statement.

C++14 decltype(auto)

Use `__has_feature(cxx_decltype_auto)` or `__has_extension(cxx_decltype_auto)` to determine if support for the `decltype(auto)` placeholder type is enabled.

C++14 default initializers for aggregates

Use `__has_feature(cxx_aggregate_nsdmi)` or `__has_extension(cxx_aggregate_nsdmi)` to determine if support for default initializers in aggregate members is enabled.

C++14 digit separators

Use `__cpp_digit_separators` to determine if support for digit separators using single quotes (for instance, `10'000`) is enabled. At this time, there is no corresponding `__has_feature` name

C++14 generalized lambda capture

Use `__has_feature(cxx_init_captures)` or `__has_extension(cxx_init_captures)` to determine if support for lambda captures with explicit initializers is enabled (for instance, `[n(0)] { return ++n; }`).

C++14 generic lambdas

Use `__has_feature(cxx_generic_lambdas)` or `__has_extension(cxx_generic_lambdas)` to determine if support for generic (polymorphic) lambdas is enabled (for instance, `[] (auto x) { return x + 1; }`).

C++14 relaxed constexpr

Use `__has_feature(cxx_relaxed_constexpr)` or `__has_extension(cxx_relaxed_constexpr)` to determine if variable declarations, local variable modification, and control flow constructs are permitted in `constexpr` functions.

C++14 return type deduction

Use `__has_feature(cxx_return_type_deduction)` or `__has_extension(cxx_return_type_deduction)` to determine if support for return type deduction for functions (using `auto` as a return type) is enabled.

C++14 runtime-sized arrays

Use `__has_feature(cxx_runtime_array)` or `__has_extension(cxx_runtime_array)` to determine if support for arrays of runtime bound (a restricted form of variable-length arrays) is enabled. Clang's implementation of this feature is incomplete.

C++14 variable templates

Use `__has_feature(cxx_variable_templates)` or `__has_extension(cxx_variable_templates)` to determine if support for templated variable declarations is enabled.

C11

The features listed below are part of the C11 standard. As a result, all these features are enabled with the `-std=c11` or `-std=gnu11` option when compiling C code. Additionally, because these features are all backward-compatible, they are available as extensions in all language modes.

C11 alignment specifiers

Use `__has_feature(c_alignas)` or `__has_extension(c_alignas)` to determine if support for alignment specifiers using `_Alignas` is enabled.

Use `__has_feature(c_alignof)` or `__has_extension(c_alignof)` to determine if support for the `_Alignof` keyword is enabled.

C11 atomic operations

Use `__has_feature(c_atomic)` or `__has_extension(c_atomic)` to determine if support for atomic types using `_Atomic` is enabled. Clang also provides a set of builtins which can be used to implement the `<stdatomic.h>` operations on `_Atomic` types. Use `__has_include(<stdatomic.h>)` to determine if C11's `<stdatomic.h>` header is available.

Clang will use the system's `<stdatomic.h>` header when one is available, and will otherwise use its own. When using its own, implementations of the atomic operations are provided as macros. In the cases where C11 also requires a real function, this header provides only the declaration of that function (along with a shadowing macro implementation), and you must link to a library which provides a definition of the function if you use it instead of the macro.

C11 generic selections

Use `__has_feature(c_generic_selections)` or `__has_extension(c_generic_selections)` to determine if support for generic selections is enabled.

As an extension, the C11 generic selection expression is available in all languages supported by Clang. The syntax is the same as that given in the C11 standard.

In C, type compatibility is decided according to the rules given in the appropriate standard, but in C++, which lacks the type compatibility rules used in C, types are considered compatible only if they are equivalent.

C11 `_Static_assert()`

Use `__has_feature(c_static_assert)` or `__has_extension(c_static_assert)` to determine if support for compile-time assertions using `_Static_assert` is enabled.

C11 `_Thread_local`

Use `__has_feature(c_thread_local)` or `__has_extension(c_thread_local)` to determine if support for `_Thread_local` variables is enabled.

Modules

Use `__has_feature(modules)` to determine if Modules have been enabled. For example, compiling code with `-fmodules` enables the use of Modules.

More information could be found here.

Type Trait Primitives

Type trait primitives are special builtin constant expressions that can be used by the standard C++ library to facilitate or simplify the implementation of user-facing type traits in the <type_traits> header.

They are not intended to be used directly by user code because they are implementation-defined and subject to change – as such they're tied closely to the supported set of system headers, currently:

- LLVM's own libc++
- GNU libstdc++
- The Microsoft standard C++ library

Clang supports the GNU C++ type traits and a subset of the Microsoft Visual C++ type traits, as well as nearly all of the Embarcadero C++ type traits.

The following type trait primitives are supported by Clang. Those traits marked (C++) provide implementations for type traits specified by the C++ standard; `__X(...)` has the same semantics and constraints as the corresponding `std::X_t<...>` or `std::X_v<...>` type trait.

- `__array_rank(type)` (Embarcadero): Returns the number of levels of array in the type `type`: `0` if `type` is not an array type, and `__array_rank(element) + 1` if `type` is an array of `element`.
- `__array_extent(type, dim)` (Embarcadero): The `dim`'th array bound in the type `type`, or `0` if `dim >= __array_rank(type)`.
- `__has_nothrow_assign` (GNU, Microsoft, Embarcadero): Deprecated, use `__is_nothrow_assignable` instead.
- `__has_nothrow_move_assign` (GNU, Microsoft): Deprecated, use `__is_nothrow_assignable` instead.
- `__has_nothrow_copy` (GNU, Microsoft): Deprecated, use `__is_nothrow_constructible` instead.
- `__has_nothrow_constructor` (GNU, Microsoft): Deprecated, use `__is_nothrow_constructible` instead.
- `__has_trivial_assign` (GNU, Microsoft, Embarcadero): Deprecated, use `__is_trivially_assignable` instead.

- `__has_trivial_move_assign` (GNU, Microsoft): Deprecated, use `__is_trivially_assignable` instead.
- `__has_trivial_copy` (GNU, Microsoft): Deprecated, use `__is_trivially_constructible` instead.
- `__has_trivial_constructor` (GNU, Microsoft): Deprecated, use `__is_trivially_constructible` instead.
- `__has_trivial_move_constructor` (GNU, Microsoft): Deprecated, use `__is_trivially_constructible` instead.
- `__has_trivial_destructor` (GNU, Microsoft, Embarcadero): Deprecated, use `__is_trivially_destructible` instead.
- `__has_unique_object_representations` (C++, GNU)
- `__has_virtual_destructor` (C++, GNU, Microsoft, Embarcadero)
- `__is_abstract` (C++, GNU, Microsoft, Embarcadero)
- `__is_aggregate` (C++, GNU, Microsoft)
- `__is_arithmetic` (C++, Embarcadero)
- `__is_array` (C++, Embarcadero)
- `__is_assignable` (C++, MSVC 2015)
- `__is_base_of` (C++, GNU, Microsoft, Embarcadero)
- `__is_class` (C++, GNU, Microsoft, Embarcadero)
- `__is_complete_type(type)` (Embarcadero): Return `true` if `type` is a complete type. Warning: this trait is dangerous because it can return different values at different points in the same program.
- `__is_compound` (C++, Embarcadero)
- `__is_const` (C++, Embarcadero)
- `__is_constructible` (C++, MSVC 2013)
- `__is_convertible` (C++, Embarcadero)
- `__is_convertible_to` (Microsoft): Synonym for `__is_convertible`.
- `__is_destructible` (C++, MSVC 2013): Only available in `-fms-extensions` mode.
- `__is_empty` (C++, GNU, Microsoft, Embarcadero)
- `__is_enum` (C++, GNU, Microsoft, Embarcadero)
- `__is_final` (C++, GNU, Microsoft)
- `__is_floating_point` (C++, Embarcadero)
- `__is_function` (C++, Embarcadero)
- `__is_fundamental` (C++, Embarcadero)
- `__is_integral` (C++, Embarcadero)
- `__is_interface_class` (Microsoft): Returns `false`, even for types defined with `__interface`.
- `__is_literal` (Clang): Synonym for `__is_literal_type`.
- `__is_literal_type` (C++, GNU, Microsoft): Note, the corresponding standard trait was deprecated in C++17 and removed in C++20.
- `__is_lvalue_reference` (C++, Embarcadero)
- `__is_member_object_pointer` (C++, Embarcadero)
- `__is_member_function_pointer` (C++, Embarcadero)
- `__is_member_pointer` (C++, Embarcadero)
- `__is_nothrow_assignable` (C++, MSVC 2013)
- `__is_nothrow_constructible` (C++, MSVC 2013)

- `__is_nothrow_destructible` (C++, MSVC 2013) Only available in `-fms-extensions` mode.
- `__is_object` (C++, Embarcadero)
- `__is_pod` (C++, GNU, Microsoft, Embarcadero): Note, the corresponding standard trait was deprecated in C++20.
- `__is_pointer` (C++, Embarcadero)
- `__is_polymorphic` (C++, GNU, Microsoft, Embarcadero)
- `__is_reference` (C++, Embarcadero)
- `__is_rvalue_reference` (C++, Embarcadero)
- `__is_same` (C++, Embarcadero)
- `__is_same_as` (GCC): Synonym for `__is_same`.
- `__is_scalar` (C++, Embarcadero)
- `__is_sealed` (Microsoft): Synonym for `__is_final`.
- `__is_signed` (C++, Embarcadero): Returns false for enumeration types, and returns true for floating-point types. Note, before Clang 10, returned true for enumeration types if the underlying type was signed, and returned false for floating-point types.
- `__is_standard_layout` (C++, GNU, Microsoft, Embarcadero)
- `__is_trivial` (C++, GNU, Microsoft, Embarcadero)
- `__is_trivially_assignable` (C++, GNU, Microsoft)
- `__is_trivially_constructible` (C++, GNU, Microsoft)
- `__is_trivially_copyable` (C++, GNU, Microsoft)
- `__is_trivially_destructible` (C++, MSVC 2013)
- `__is_trivially_relocatable` (Clang): Returns true if moving an object of the given type, and then destroying the source object, is known to be functionally equivalent to copying the underlying bytes and then dropping the source object on the floor. This is true of trivial types and types which were made trivially relocatable via the `clang::trivial_abi` attribute.
- `__is_union` (C++, GNU, Microsoft, Embarcadero)
- `__is_unsigned` (C++, Embarcadero): Returns false for enumeration types. Note, before Clang 13, returned true for enumeration types if the underlying type was unsigned.
- `__is_void` (C++, Embarcadero)
- `__is_volatile` (C++, Embarcadero)
- `__reference_binds_to_temporary(T, U)` (Clang): Determines whether a reference of type `T` bound to an expression of type `U` would bind to a materialized temporary object. If `T` is not a reference type the result is false. Note this trait will also return false when the initialization of `T` from `U` is ill-formed.
- `__underlying_type` (C++, GNU, Microsoft)

In addition, the following expression traits are supported:

- `__is_lvalue_expr(e)` (Embarcadero): Returns true if `e` is an lvalue expression. Deprecated, use `__is_lvalue_reference(decltype((e)))` instead.
- `__is_rvalue_expr(e)` (Embarcadero): Returns true if `e` is a prvalue expression. Deprecated, use `!__is_reference(decltype((e)))` instead.

There are multiple ways to detect support for a type trait `__X` in the compiler, depending on the oldest version of Clang you wish to support.

- From Clang 10 onwards, `__has_builtin(__X)` can be used.
- From Clang 6 onwards, `!__is_identifier(__X)` can be used.
- From Clang 3 onwards, `__has_feature(X)` can be used, but only supports the following traits:
    - `__has_nothrow_assign`

- `__has_nothrow_copy`
- `__has_nothrow_constructor`
- `__has_trivial_assign`
- `__has_trivial_copy`
- `__has_trivial_constructor`
- `__has_trivial_destructor`
- `__has_virtual_destructor`
- `__is_abstract`
- `__is_base_of`
- `__is_class`
- `__is_constructible`
- `__is_convertible_to`
- `__is_empty`
- `__is_enum`
- `__is_final`
- `__is_literal`
- `__is_standard_layout`
- `__is_pod`
- `__is_polymorphic`
- `__is_sealed`
- `__is_trivial`
- `__is_trivially_assignable`
- `__is_trivially_constructible`
- `__is_trivially_copyable`
- `__is_union`
- `__underlying_type`

A simplistic usage example as might be seen in standard C++ headers follows:

```
#if __has_builtin(__is_convertible_to)
template<typename From, typename To>
struct is_convertible_to {
  static const bool value = __is_convertible_to(From, To);
};
#else
// Emulate type trait for compatibility with other compilers.
#endif
```

Blocks

The syntax and high level language feature description is in BlockLanguageSpec. Implementation and ABI details for the clang implementation are in Block-ABI-Apple.

Query for this feature with `__has_extension(blocks)`.

ASM Goto with Output Constraints

In addition to the functionality provided by GCC's extended assembly, clang supports output constraints with the *goto* form.

The goto form of GCC's extended assembly allows the programmer to branch to a C label from within an inline assembly block. Clang extends this behavior by allowing the programmer to use output constraints:

```
int foo(int x) {
    int y;
    asm goto("# %0 %1 %l2" : "=r"(y) : "r"(x) : : err);
    return y;
  err:
    return -1;
}
```

It's important to note that outputs are valid only on the "fallthrough" branch. Using outputs on an indirect branch may result in undefined behavior. For example, in the function above, use of the value assigned to *y* in the *err* block is undefined behavior.

When using tied-outputs (i.e. outputs that are inputs and outputs, not just outputs) with the *+r* constraint, there is a hidden input that's created before the label, so numeric references to operands must account for that.

```
int foo(int x) {
    // %0 and %1 both refer to x
    // %l2 refers to err
    asm goto("# %0 %1 %l2" : "+r"(x) : : : err);
    return x;
  err:
    return -1;
}
```

This was changed to match GCC in clang-13; for better portability, symbolic references can be used instead of numeric references.

```
int foo(int x) {
    asm goto("# %[x] %l[err]" : [x]"+r"(x) : : : err);
    return x;
  err:
    return -1;
}
```

Query for this feature with `__has_extension(gnu_asm_goto_with_outputs)`.

Objective-C Features

Related result types

According to Cocoa conventions, Objective-C methods with certain names ("`init`", "`alloc`", etc.) always return objects that are an instance of the receiving class's type. Such methods are said to have a "related result type", meaning that a message send to one of these methods will have the same static type as an instance of the receiver class. For example, given the following classes:

```
@interface NSObject
+ (id)alloc;
- (id)init;
@end

@interface NSArray : NSObject
@end
```

and this common initialization pattern

```
NSArray *array = [[NSArray alloc] init];
```

the type of the expression `[NSArray alloc]` is `NSArray*` because `alloc` implicitly has a related result type. Similarly, the type of the expression `[[NSArray alloc] init]` is `NSArray*`, since `init` has a related result type and its receiver is known to have the type `NSArray *`. If neither `alloc` nor `init` had a related result type, the expressions would have had type `id`, as declared in the method signature.

A method with a related result type can be declared by using the type `instancetype` as its result type. `instancetype` is a contextual keyword that is only permitted in the result type of an Objective-C method, e.g.

```
@interface A
+ (instancetype)constructAnA;
@end
```

The related result type can also be inferred for some methods. To determine whether a method has an inferred related result type, the first word in the camel-case selector (e.g., "`init`" in "`initWithObjects`") is considered, and the method will have a related result type if its return type is compatible with the type of its class and if:

- the first word is "`alloc`" or "`new`", and the method is a class method, or
- the first word is "`autorelease`", "`init`", "`retain`", or "`self`", and the method is an instance method.

If a method with a related result type is overridden by a subclass method, the subclass method must also return a type that is compatible with the subclass type. For example:

```
@interface NSString : NSObject
- (NSUnrelated *)init; // incorrect usage: NSUnrelated is not NSString or a superclass of NSString
@end
```

Related result types only affect the type of a message send or property access via the given method. In all other respects, a method with a related result type is treated the same way as method that returns `id`.

Use `__has_feature(objc_instancetype)` to determine whether the `instancetype` contextual keyword is available.

Automatic reference counting

Clang provides support for automated reference counting in Objective-C, which eliminates the need for manual `retain`/`release`/`autorelease` message sends. There are three feature macros associated with automatic reference counting: `__has_feature(objc_arc)` indicates the availability of automated reference counting in general, while `__has_feature(objc_arc_weak)` indicates that automated reference counting also includes support for `__weak` pointers to Objective-C objects. `__has_feature(objc_arc_fields)` indicates that C structs are allowed to have fields that are pointers to Objective-C objects managed by automatic reference counting.

Weak references

Clang supports ARC-style weak and unsafe references in Objective-C even outside of ARC mode. Weak references must be explicitly enabled with the `-fobjc-weak` option; use `__has_feature((objc_arc_weak))` to test whether they are enabled. Unsafe references are enabled unconditionally. ARC-style weak and unsafe references cannot be used when Objective-C garbage collection is enabled.

Except as noted below, the language rules for the `__weak` and `__unsafe_unretained` qualifiers (and the `weak` and `unsafe_unretained` property attributes) are just as laid out in the ARC specification. In particular, note that some classes do not support forming weak references to their instances, and note that special care must be taken when storing weak references in memory where initialization and deinitialization are outside the responsibility of the compiler (such as in `malloc`-ed memory).

Loading from a `__weak` variable always implicitly retains the loaded value. In non-ARC modes, this retain is normally balanced by an implicit autorelease. This autorelease can be suppressed by performing the load in the receiver position of a `-retain` message send (e.g. `[weakReference retain]`); note that this performs only a single retain (the retain done when primitively loading from the weak reference).

For the most part, `__unsafe_unretained` in non-ARC modes is just the default behavior of variables and therefore is not needed. However, it does have an effect on the semantics of block captures: normally, copying a block which captures an Objective-C object or block pointer causes the captured pointer to be retained or copied, respectively, but that behavior is suppressed when the captured variable is qualified with `__unsafe_unretained`.

Note that the `__weak` qualifier formerly meant the GC qualifier in all non-ARC modes and was silently ignored outside of GC modes. It now means the ARC-style qualifier in all non-GC modes and is no longer allowed if not enabled by either `-fobjc-arc` or `-fobjc-weak`. It is expected that `-fobjc-weak` will eventually be enabled by default in all non-GC Objective-C modes.

Enumerations with a fixed underlying type

Clang provides support for C++11 enumerations with a fixed underlying type within Objective-C. For example, one can write an enumeration type as:

```
typedef enum : unsigned char { Red, Green, Blue } Color;
```

This specifies that the underlying type, which is used to store the enumeration value, is `unsigned char`.

Use `__has_feature(objc_fixed_enum)` to determine whether support for fixed underlying types is available in Objective-C.

Interoperability with C++11 lambdas

Clang provides interoperability between C++11 lambdas and blocks-based APIs, by permitting a lambda to be implicitly converted to a block pointer with the corresponding signature. For example, consider an API such as `NSArray`'s array-sorting method:

```
- (NSArray *)sortedArrayUsingComparator:(NSComparator)cmptr;
```

`NSComparator` is simply a typedef for the block pointer `NSComparisonResult (^)(id, id)`, and parameters of this type are generally provided with block literals as arguments. However, one can also use a C++11 lambda so long as it provides the same signature (in this case, accepting two parameters of type `id` and returning an `NSComparisonResult`):

```
NSArray *array = @[@"string 1", @"string 21", @"string 12", @"String 11",
                   @"String 02"];
const NSStringCompareOptions comparisonOptions
  = NSCaseInsensitiveSearch | NSNumericSearch |
    NSWidthInsensitiveSearch | NSForcedOrderingSearch;
NSLocale *currentLocale = [NSLocale currentLocale];
NSArray *sorted
  = [array sortedArrayUsingComparator:[=](id s1, id s2) -> NSComparisonResult {
            NSRange string1Range = NSMakeRange(0, [s1 length]);
            return [s1 compare:s2 options:comparisonOptions
            range:string1Range locale:currentLocale];
     }];
NSLog(@"sorted: %@", sorted);
```

This code relies on an implicit conversion from the type of the lambda expression (an unnamed, local class type called the *closure type*) to the corresponding block pointer type. The conversion itself is expressed by a conversion operator in that closure type that produces a block pointer with the same signature as the lambda itself, e.g.,

```
operator NSComparisonResult (^)(id, id)() const;
```

This conversion function returns a new block that simply forwards the two parameters to the lambda object (which it captures by copy), then returns the result. The returned block is first copied (with `Block_copy`) and then autoreleased. As an optimization, if a lambda expression is immediately converted to a block pointer (as in the first example, above), then the block is not copied and autoreleased: rather, it is given the same lifetime as a block literal written at that point in the program, which avoids the overhead of copying a block to the heap in the common case.

The conversion from a lambda to a block pointer is only available in Objective-C++, and not in C++ with blocks, due to its use of Objective-C memory management (autorelease).

Object Literals and Subscripting

Clang provides support for Object Literals and Subscripting in Objective-C, which simplifies common Objective-C programming patterns, makes programs more concise, and improves the safety of container creation. There are several feature macros associated with object literals and subscripting: `__has_feature(objc_array_literals)` tests the availability of array literals; `__has_feature(objc_dictionary_literals)` tests the availability of dictionary literals; `__has_feature(objc_subscripting)` tests the availability of object subscripting.

Objective-C Autosynthesis of Properties

Clang provides support for autosynthesis of declared properties. Using this feature, clang provides default synthesis of those properties not declared @dynamic and not having user provided backing getter and setter methods. `__has_feature(objc_default_synthesize_properties)` checks for availability of this feature in version of clang being used.

Objective-C retaining behavior attributes

In Objective-C, functions and methods are generally assumed to follow the Cocoa Memory Management conventions for ownership of object arguments and return values. However, there are exceptions, and so Clang provides attributes to allow these exceptions to be documented. This are used by ARC and the static analyzer Some exceptions may be better described using the `objc_method_family` attribute instead.

Usage: The `ns_returns_retained`, `ns_returns_not_retained`, `ns_returns_autoreleased`, `cf_returns_retained`, and `cf_returns_not_retained` attributes can be placed on methods and functions that return Objective-C or CoreFoundation objects. They are commonly placed at the end of a function prototype or method declaration:

```
id foo() __attribute__((ns_returns_retained));

- (NSString *)bar:(int)x __attribute__((ns_returns_retained));
```

The `*_returns_retained` attributes specify that the returned object has a +1 retain count. The `*_returns_not_retained` attributes specify that the return object has a +0 retain count, even if the normal convention for its selector would be +1. `ns_returns_autoreleased` specifies that the returned object is +0, but is guaranteed to live at least as long as the next flush of an autorelease pool.

Usage: The `ns_consumed` and `cf_consumed` attributes can be placed on an parameter declaration; they specify that the argument is expected to have a +1 retain count, which will be balanced in some way by the function or method. The `ns_consumes_self` attribute can only be placed on an Objective-C method; it specifies that the method expects its `self` parameter to have a +1 retain count, which it will balance in some way.

```
void foo(__attribute__((ns_consumed)) NSString *string);

- (void) bar __attribute__((ns_consumes_self));
- (void) baz:(id) __attribute__((ns_consumed)) x;
```

Further examples of these attributes are available in the static analyzer's list of annotations for analysis.

Query for these features with `__has_attribute(ns_consumed)`, `__has_attribute(ns_returns_retained)`, etc.

Objective-C @available

It is possible to use the newest SDK but still build a program that can run on older versions of macOS and iOS by passing `-mmacosx-version-min=` / `-miphoneos-version-min=`.

Before LLVM 5.0, when calling a function that exists only in the OS that's newer than the target OS (as determined by the minimum deployment version), programmers had to carefully check if the function exists at runtime, using null checks for weakly-linked C functions, `+class` for Objective-C classes, and `-respondsToSelector:` or `+instancesRespondToSelector:` for Objective-C methods. If such a check was missed, the program would compile fine, run fine on newer systems, but crash on older systems.

As of LLVM 5.0, `-Wunguarded-availability` uses the availability attributes together with the new `@available()` keyword to assist with this issue. When a method that's introduced in the OS newer than the target OS is called, a -Wunguarded-availability warning is emitted if that call is not guarded:

```
void my_fun(NSSomeClass* var) {
  // If fancyNewMethod was added in e.g. macOS 10.12, but the code is
  // built with -mmacosx-version-min=10.11, then this unconditional call
  // will emit a -Wunguarded-availability warning:
  [var fancyNewMethod];
}
```

To fix the warning and to avoid the crash on macOS 10.11, wrap it in `if(@available())`:

```
void my_fun(NSSomeClass* var) {
  if (@available(macOS 10.12, *)) {
    [var fancyNewMethod];
  } else {
    // Put fallback behavior for old macOS versions (and for non-mac
    // platforms) here.
  }
}
```

The * is required and means that platforms not explicitly listed will take the true branch, and the compiler will emit `-Wunguarded-availability` warnings for unlisted platforms based on those platform's deployment target. More than one platform can be listed in `@available()`:

```
void my_fun(NSSomeClass* var) {
  if (@available(macOS 10.12, iOS 10, *)) {
    [var fancyNewMethod];
  }
}
```

If the caller of `my_fun()` already checks that `my_fun()` is only called on 10.12, then add an availability attribute to it, which will also suppress the warning and require that calls to my_fun() are checked:

```
API_AVAILABLE(macos(10.12)) void my_fun(NSSomeClass* var) {
  [var fancyNewMethod];  // Now ok.
}
```

`@available()` is only available in Objective-C code. To use the feature in C and C++ code, use the `__builtin_available()` spelling instead.

If existing code uses null checks or `-respondsToSelector:`, it should be changed to use `@available()` (or `__builtin_available`) instead.

`-Wunguarded-availability` is disabled by default, but `-Wunguarded-availability-new`, which only emits this warning for APIs that have been introduced in macOS >= 10.13, iOS >= 11, watchOS >= 4 and tvOS >= 11, is enabled by default.

Objective-C++ ABI: protocol-qualifier mangling of parameters

Starting with LLVM 3.4, Clang produces a new mangling for parameters whose type is a qualified-`id` (e.g., `id<Foo>`). This mangling allows such parameters to be differentiated from those with the regular unqualified `id` type.

This was a non-backward compatible mangling change to the ABI. This change allows proper overloading, and also prevents mangling conflicts with template parameters of protocol-qualified type.

Query the presence of this new mangling with `__has_feature(objc_protocol_qualifier_mangling)`.

Initializer lists for complex numbers in C

clang supports an extension which allows the following in C:

```
#include <math.h>
#include <complex.h>
complex float x = { 1.0f, INFINITY }; // Init to (1, Inf)
```

This construct is useful because there is no way to separately initialize the real and imaginary parts of a complex variable in standard C, given that clang does not support `_Imaginary`. (Clang also supports the `__real__` and `__imag__` extensions from gcc, which help in some cases, but are not usable in static initializers.)

Note that this extension does not allow eliding the braces; the meaning of the following two lines is different:

```
complex float x[] = { { 1.0f, 1.0f } }; // [0] = (1, 1)
complex float x[] = { 1.0f, 1.0f }; // [0] = (1, 0), [1] = (1, 0)
```

This extension also works in C++ mode, as far as that goes, but does not apply to the C++ `std::complex`. (In C++11, list initialization allows the same syntax to be used with `std::complex` with the same meaning.)

For GCC compatibility, `__builtin_complex(re, im)` can also be used to construct a complex number from the given real and imaginary components.

OpenCL Features

Clang supports internal OpenCL extensions documented below.

`__cl_clang_bitfields`

With this extension it is possible to enable bitfields in structs or unions using the OpenCL extension pragma mechanism detailed in the OpenCL Extension Specification, section 1.2.

Use of bitfields in OpenCL kernels can result in reduced portability as struct layout is not guaranteed to be consistent when compiled by different compilers. If structs with bitfields are used as kernel function parameters, it can result in incorrect functionality when the layout is different between the host and device code.

Example of Use:

```
#pragma OPENCL EXTENSION __cl_clang_bitfields : enable
struct with_bitfield {
  unsigned int i : 5; // compiled - no diagnostic generated
};

#pragma OPENCL EXTENSION __cl_clang_bitfields : disable
struct without_bitfield {
  unsigned int i : 5; // error - bitfields are not supported
};
```

`__cl_clang_function_pointers`

With this extension it is possible to enable various language features that are relying on function pointers using regular OpenCL extension pragma mechanism detailed in the OpenCL Extension Specification, section 1.2.

In C++ for OpenCL this also enables:

- Use of member function pointers;
- Unrestricted use of references to functions;
- Virtual member functions.

Such functionality is not conformant and does not guarantee to compile correctly in any circumstances. It can be used if:

- the kernel source does not contain call expressions to (member-) function pointers, or virtual functions. For example this extension can be used in metaprogramming algorithms to be able to specify/detect types generically.
- the generated kernel binary does not contain indirect calls because they are eliminated using compiler optimizations e.g. devirtualization.
- the selected target supports the function pointer like functionality e.g. most CPU targets.

Example of Use:

```
#pragma OPENCL EXTENSION __cl_clang_function_pointers : enable
void foo()
{
  void (*fp)(); // compiled - no diagnostic generated
}

#pragma OPENCL EXTENSION __cl_clang_function_pointers : disable
void bar()
{
  void (*fp)(); // error - pointers to function are not allowed
}
```

__cl_clang_variadic_functions

With this extension it is possible to enable variadic arguments in functions using regular OpenCL extension pragma mechanism detailed in the OpenCL Extension Specification, section 1.2.

This is not conformant behavior and it can only be used portably when the functions with variadic prototypes do not get generated in binary e.g. the variadic prototype is used to specify a function type with any number of arguments in metaprogramming algorithms in C++ for OpenCL.

This extensions can also be used when the kernel code is intended for targets supporting the variadic arguments e.g. majority of CPU targets.

Example of Use:

```
#pragma OPENCL EXTENSION __cl_clang_variadic_functions : enable
void foo(int a, ...); // compiled - no diagnostic generated

#pragma OPENCL EXTENSION __cl_clang_variadic_functions : disable
void bar(int a, ...); // error - variadic prototype is not allowed
```

__cl_clang_non_portable_kernel_param_types

With this extension it is possible to enable the use of some restricted types in kernel parameters specified in C++ for OpenCL v1.0 s2.4. The restrictions can be relaxed using regular OpenCL extension pragma mechanism detailed in the OpenCL Extension Specification, section 1.2.

This is not a conformant behavior and it can only be used when the kernel arguments are not accessed on the host side or the data layout/size between the host and device is known to be compatible.

Example of Use:

```
// Plain Old Data type.
struct Pod {
  int a;
  int b;
};

// Not POD type because of the constructor.
// Standard layout type because there is only one access control.
struct OnlySL {
  int a;
  int b;
  NotPod() : a(0), b(0) {}
};

// Not standard layout type because of two different access controls.
struct NotSL {
  int a;
private:
  int b;
}

kernel void kernel_main(
  Pod a,
#pragma OPENCL EXTENSION __cl_clang_non_portable_kernel_param_types : enable
  OnlySL b,
  global NotSL *c,
#pragma OPENCL EXTENSION __cl_clang_non_portable_kernel_param_types : disable
  global OnlySL *d,
);
```

Remove address space builtin function

`__remove_address_space` allows to derive types in C++ for OpenCL that have address space qualifiers removed. This utility only affects address space qualifiers, therefore, other type qualifiers such as `const` or `volatile` remain unchanged.

Example of Use:

```
template<typename T>
void foo(T *par){
  T var1; // error - local function variable with global address space
  __private T var2; // error - conflicting address space qualifiers
  __private __remove_address_space<T>::type var3; // var3 is __private int
}

void bar(){
  __global int* ptr;
  foo(ptr);
}
```

Legacy 1.x atomics with generic address space

Clang allows use of atomic functions from the OpenCL 1.x standards with the generic address space pointer in C++ for OpenCL mode.

This is a non-portable feature and might not be supported by all targets.

Example of Use:

```
void foo(__generic volatile unsigned int* a) {
  atomic_add(a, 1);
}
```

Builtin Functions

Clang supports a number of builtin library functions with the same syntax as GCC, including things like `__builtin_nan`, `__builtin_constant_p`, `__builtin_choose_expr`, `__builtin_types_compatible_p`, `__builtin_assume_aligned`, `__sync_fetch_and_add`, etc. In addition to the GCC builtins, Clang supports a number of builtins that GCC does not, which are listed here.

Please note that Clang does not and will not support all of the GCC builtins for vector operations. Instead of using builtins, you should use the functions defined in target-specific header files like `<xmmintrin.h>`, which define portable wrappers for these. Many of the Clang versions of these functions are implemented directly in terms of extended vector support instead of builtins, in order to reduce the number of builtins that we need to implement.

`__builtin_alloca`

`__builtin_alloca` is used to dynamically allocate memory on the stack. Memory is automatically freed upon function termination.

Syntax:

```
__builtin_alloca(size_t n)
```

Example of Use:

```
void init(float* data, size_t nbelems);
void process(float* data, size_t nbelems);
int foo(size_t n) {
  auto mem = (float*)__builtin_alloca(n * sizeof(float));
  init(mem, n);
  process(mem, n);
  /* mem is automatically freed at this point */
}
```

Description:

`__builtin_alloca` is meant to be used to allocate a dynamic amount of memory on the stack. This amount is subject to stack allocation limits.

Query for this feature with `__has_builtin(__builtin_alloca)`.

`__builtin_alloca_with_align`

`__builtin_alloca_with_align` is used to dynamically allocate memory on the stack while controlling its alignment. Memory is automatically freed upon function termination.

Syntax:

```
__builtin_alloca_with_align(size_t n, size_t align)
```

Example of Use:

```
void init(float* data, size_t nbelems);
void process(float* data, size_t nbelems);
int foo(size_t n) {
  auto mem = (float*)__builtin_alloca_with_align(
                     n * sizeof(float),
                     CHAR_BIT * alignof(float));
  init(mem, n);
  process(mem, n);
  /* mem is automatically freed at this point */
}
```

Description:

`__builtin_alloca_with_align` is meant to be used to allocate a dynamic amount of memory on the stack. It is similar to `__builtin_alloca` but accepts a second argument whose value is the alignment constraint, as a power of 2 in *bits*.

Query for this feature with `__has_builtin(__builtin_alloca_with_align)`.

`__builtin_assume`

`__builtin_assume` is used to provide the optimizer with a boolean invariant that is defined to be true.

Syntax:

```
__builtin_assume(bool)
```

Example of Use:

```
int foo(int x) {
    __builtin_assume(x != 0);
    // The optimizer may short-circuit this check using the invariant.
    if (x == 0)
          return do_something();
    return do_something_else();
}
```

Description:

The boolean argument to this function is defined to be true. The optimizer may analyze the form of the expression provided as the argument and deduce from that information used to optimize the program. If the condition is violated during execution, the behavior is undefined. The argument itself is never evaluated, so any side effects of the expression will be discarded.

Query for this feature with `__has_builtin(__builtin_assume)`.

__builtin_call_with_static_chain

__builtin_call_with_static_chain is used to perform a static call while setting updating the static chain register.

Syntax:

```
T __builtin_call_with_static_chain(T expr, void* ptr)
```

Example of Use:

```
auto v = __builtin_call_with_static_chain(foo(3), foo);
```

Description:

This builtin returns `expr` after checking that `expr` is a non-member static call expression. The call to that expression is made while using `ptr` as a function pointer stored in a dedicated register to implement *static chain* calling convention, as used by some language to implement closures or nested functions.

Query for this feature with __has_builtin(__builtin_call_with_static_chain).

__builtin_readcyclecounter

__builtin_readcyclecounter is used to access the cycle counter register (or a similar low-latency, high-accuracy clock) on those targets that support it.

Syntax:

```
__builtin_readcyclecounter()
```

Example of Use:

```
unsigned long long t0 = __builtin_readcyclecounter();
do_something();
unsigned long long t1 = __builtin_readcyclecounter();
unsigned long long cycles_to_do_something = t1 - t0; // assuming no overflow
```

Description:

The __builtin_readcyclecounter() builtin returns the cycle counter value, which may be either global or process/thread-specific depending on the target. As the backing counters often overflow quickly (on the order of seconds) this should only be used for timing small intervals. When not supported by the target, the return value is always zero. This builtin takes no arguments and produces an unsigned long long result.

Query for this feature with __has_builtin(__builtin_readcyclecounter). Note that even if present, its use may depend on run-time privilege or other OS controlled state.

__builtin_dump_struct

Syntax:

```
__builtin_dump_struct(&some_struct, some_printf_func, args...);
```

Examples:

```
struct S {
  int x, y;
  float f;
  struct T {
    int i;
  } t;
};

void func(struct S *s) {
  __builtin_dump_struct(s, printf);
}
```

Example output:

```
struct S {
  int x = 100
  int y = 42
  float f = 3.141593
  struct T t = {
    int i = 1997
  }
}

#include <string>
struct T { int a, b; };
constexpr void constexpr_sprintf(std::string &out, const char *format,
                                 auto ...args) {
  // ...
}
constexpr std::string dump_struct(auto &x) {
  std::string s;
  __builtin_dump_struct(&x, constexpr_sprintf, s);
  return s;
}
static_assert(dump_struct(T{1, 2}) == R"(struct T {
  int a = 1
  int b = 2
}
)");
```

Description:

The `__builtin_dump_struct` function is used to print the fields of a simple structure and their values for debugging purposes. The first argument of the builtin should be a pointer to the struct to dump. The second argument `f` should be some callable expression, and can be a function object or an overload set. The builtin calls `f`, passing any further arguments `args...` followed by a `printf`-compatible format string and the corresponding arguments. `f` may be called more than once, and `f` and `args` will be evaluated once per call. In C++, `f` may be a template or overload set and resolve to different functions for each call.

In the format string, a suitable format specifier will be used for builtin types that Clang knows how to format. This includes standard builtin types, as well as aggregate structures, `void*` (printed with %p), and `const char*` (printed with %s). A *%p specifier will be used for a field that Clang doesn't know how to format, and the corresopnding argument will be a pointer to the field. This allows a C++ templated formatting function to detect this case and implement custom formatting. A * will otherwise not precede a format specifier.

This builtin does not return a value.

This builtin can be used in constant expressions.

Query for this feature with `__has_builtin(__builtin_dump_struct)`

`__builtin_shufflevector`

`__builtin_shufflevector` is used to express generic vector permutation/shuffle/swizzle operations. This builtin is also very important for the implementation of various target-specific header files like `<xmmintrin.h>`.

Syntax:

```
__builtin_shufflevector(vec1, vec2, index1, index2, ...)
```

Examples:

```
// identity operation - return 4-element vector v1.
__builtin_shufflevector(v1, v1, 0, 1, 2, 3)

// "Splat" element 0 of V1 into a 4-element result.
__builtin_shufflevector(V1, V1, 0, 0, 0, 0)
```

```
// Reverse 4-element vector V1.
__builtin_shufflevector(V1, V1, 3, 2, 1, 0)

// Concatenate every other element of 4-element vectors V1 and V2.
__builtin_shufflevector(V1, V2, 0, 2, 4, 6)

// Concatenate every other element of 8-element vectors V1 and V2.
__builtin_shufflevector(V1, V2, 0, 2, 4, 6, 8, 10, 12, 14)

// Shuffle v1 with some elements being undefined
__builtin_shufflevector(v1, v1, 3, -1, 1, -1)
```

Description:

The first two arguments to `__builtin_shufflevector` are vectors that have the same element type. The remaining arguments are a list of integers that specify the elements indices of the first two vectors that should be extracted and returned in a new vector. These element indices are numbered sequentially starting with the first vector, continuing into the second vector. Thus, if `vec1` is a 4-element vector, index 5 would refer to the second element of `vec2`. An index of -1 can be used to indicate that the corresponding element in the returned vector is a don't care and can be optimized by the backend.

The result of `__builtin_shufflevector` is a vector with the same element type as `vec1`/`vec2` but that has an element count equal to the number of indices specified.

Query for this feature with `__has_builtin(__builtin_shufflevector)`.

`__builtin_convertvector`

`__builtin_convertvector` is used to express generic vector type-conversion operations. The input vector and the output vector type must have the same number of elements.

Syntax:

```
__builtin_convertvector(src_vec, dst_vec_type)
```

Examples:

```
typedef double vector4double __attribute__((__vector_size__(32)));
typedef float  vector4float  __attribute__((__vector_size__(16)));
typedef short  vector4short  __attribute__((__vector_size__(8)));
vector4float vf; vector4short vs;

// convert from a vector of 4 floats to a vector of 4 doubles.
__builtin_convertvector(vf, vector4double)
// equivalent to:
(vector4double) { (double) vf[0], (double) vf[1], (double) vf[2], (double) vf[3] }

// convert from a vector of 4 shorts to a vector of 4 floats.
__builtin_convertvector(vs, vector4float)
// equivalent to:
(vector4float) { (float) vs[0], (float) vs[1], (float) vs[2], (float) vs[3] }
```

Description:

The first argument to `__builtin_convertvector` is a vector, and the second argument is a vector type with the same number of elements as the first argument.

The result of `__builtin_convertvector` is a vector with the same element type as the second argument, with a value defined in terms of the action of a C-style cast applied to each element of the first argument.

Query for this feature with `__has_builtin(__builtin_convertvector)`.

`__builtin_bitreverse`

- `__builtin_bitreverse8`

- `__builtin_bitreverse16`
- `__builtin_bitreverse32`
- `__builtin_bitreverse64`

Syntax:

```
__builtin_bitreverse32(x)
```

Examples:

```
uint8_t rev_x = __builtin_bitreverse8(x);
uint16_t rev_x = __builtin_bitreverse16(x);
uint32_t rev_y = __builtin_bitreverse32(y);
uint64_t rev_z = __builtin_bitreverse64(z);
```

Description:

The '`__builtin_bitreverse`' family of builtins is used to reverse the bitpattern of an integer value; for example `0b10110110` becomes `0b01101101`. These builtins can be used within constant expressions.

`__builtin_rotateleft`

- `__builtin_rotateleft8`
- `__builtin_rotateleft16`
- `__builtin_rotateleft32`
- `__builtin_rotateleft64`

Syntax:

```
__builtin_rotateleft32(x, y)
```

Examples:

```
uint8_t rot_x = __builtin_rotateleft8(x, y);
uint16_t rot_x = __builtin_rotateleft16(x, y);
uint32_t rot_x = __builtin_rotateleft32(x, y);
uint64_t rot_x = __builtin_rotateleft64(x, y);
```

Description:

The '`__builtin_rotateleft`' family of builtins is used to rotate the bits in the first argument by the amount in the second argument. For example, `0b10000110` rotated left by 11 becomes `0b00110100`. The shift value is treated as an unsigned amount modulo the size of the arguments. Both arguments and the result have the bitwidth specified by the name of the builtin. These builtins can be used within constant expressions.

`__builtin_rotateright`

- `__builtin_rotateright8`
- `__builtin_rotateright16`
- `__builtin_rotateright32`
- `__builtin_rotateright64`

Syntax:

```
__builtin_rotateright32(x, y)
```

Examples:

```
uint8_t rot_x = __builtin_rotateright8(x, y);
uint16_t rot_x = __builtin_rotateright16(x, y);
uint32_t rot_x = __builtin_rotateright32(x, y);
uint64_t rot_x = __builtin_rotateright64(x, y);
```

Description:

The '`__builtin_rotateright`' family of builtins is used to rotate the bits in the first argument by the amount in the second argument. For example, `0b10000110` rotated right by 3 becomes `0b11010000`. The shift value is treated as an unsigned amount modulo the size of the arguments. Both arguments and the result have the bitwidth specified by the name of the builtin. These builtins can be used within constant expressions.

`__builtin_unreachable`

`__builtin_unreachable` is used to indicate that a specific point in the program cannot be reached, even if the compiler might otherwise think it can. This is useful to improve optimization and eliminates certain warnings. For example, without the `__builtin_unreachable` in the example below, the compiler assumes that the inline asm can fall through and prints a "function declared '`noreturn`' should not return" warning.

Syntax:

```
__builtin_unreachable()
```

Example of use:

```
void myabort(void) __attribute__((noreturn));
void myabort(void) {
  asm("int3");
  __builtin_unreachable();
}
```

Description:

The `__builtin_unreachable()` builtin has completely undefined behavior. Since it has undefined behavior, it is a statement that it is never reached and the optimizer can take advantage of this to produce better code. This builtin takes no arguments and produces a void result.

Query for this feature with `__has_builtin(__builtin_unreachable)`.

`__builtin_unpredictable`

`__builtin_unpredictable` is used to indicate that a branch condition is unpredictable by hardware mechanisms such as branch prediction logic.

Syntax:

```
__builtin_unpredictable(long long)
```

Example of use:

```
if (__builtin_unpredictable(x > 0)) {
   foo();
}
```

Description:

The `__builtin_unpredictable()` builtin is expected to be used with control flow conditions such as in `if` and `switch` statements.

Query for this feature with `__has_builtin(__builtin_unpredictable)`.

`__builtin_expect`

`__builtin_expect` is used to indicate that the value of an expression is anticipated to be the same as a statically known result.

Syntax:

```
long __builtin_expect(long expr, long val)
```

Example of use:

```
if (__builtin_expect(x, 0)) {
   bar();
}
```

Description:

The `__builtin_expect()` builtin is typically used with control flow conditions such as in `if` and `switch` statements to help branch prediction. It means that its first argument `expr` is expected to take the value of its second argument `val`. It always returns `expr`.

Query for this feature with `__has_builtin(__builtin_expect)`.

`__builtin_expect_with_probability`

`__builtin_expect_with_probability` is similar to `__builtin_expect` but it takes a probability as third argument.

Syntax:

```
long __builtin_expect_with_probability(long expr, long val, double p)
```

Example of use:

```
if (__builtin_expect_with_probability(x, 0, .3)) {
   bar();
}
```

Description:

The `__builtin_expect_with_probability()` builtin is typically used with control flow conditions such as in `if` and `switch` statements to help branch prediction. It means that its first argument `expr` is expected to take the value of its second argument `val` with probability `p`. `p` must be within `[0.0 ; 1.0]` bounds. This builtin always returns the value of `expr`.

Query for this feature with `__has_builtin(__builtin_expect_with_probability)`.

`__builtin_prefetch`

`__builtin_prefetch` is used to communicate with the cache handler to bring data into the cache before it gets used.

Syntax:

```
void __builtin_prefetch(const void *addr, int rw=0, int locality=3)
```

Example of use:

```
__builtin_prefetch(a + i);
```

Description:

The `__builtin_prefetch(addr, rw, locality)` builtin is expected to be used to avoid cache misses when the developper has a good understanding of which data are going to be used next. `addr` is the address that needs to be brought into the cache. `rw` indicates the expected access mode: `0` for *read* and `1` for *write*. In case of *read write* access, `1` is to be used. `locality` indicates the expected persistance of data in cache, from `0` which means that data can be discarded from cache after its next use to 3 which means that data is going to be reused a lot once in cache. 1 and 2 provide intermediate behavior between these two extremes.

Query for this feature with `__has_builtin(__builtin_prefetch)`.

`__sync_swap`

`__sync_swap` is used to atomically swap integers or pointers in memory.

Syntax:

```
type __sync_swap(type *ptr, type value, ...)
```

Example of Use:

```
int old_value = __sync_swap(&value, new_value);
```

Description:

The `__sync_swap()` builtin extends the existing `__sync_*()` family of atomic intrinsics to allow code to atomically swap the current value with the new value. More importantly, it helps developers write more efficient and correct code by avoiding expensive loops around `__sync_bool_compare_and_swap()` or relying on the platform specific implementation details of `__sync_lock_test_and_set()`. The `__sync_swap()` builtin is a full barrier.

`__builtin_addressof`

`__builtin_addressof` performs the functionality of the built-in `&` operator, ignoring any `operator&` overload. This is useful in constant expressions in C++11, where there is no other way to take the address of an object that overloads `operator&`.

Example of use:

```
template<typename T> constexpr T *addressof(T &value) {
  return __builtin_addressof(value);
}
```

`__builtin_function_start`

`__builtin_function_start` returns the address of a function body.

Syntax:

```
void *__builtin_function_start(function)
```

Example of use:

```
void a() {}
void *p = __builtin_function_start(a);

class A {
public:
  void a(int n);
  void a();
};

void A::a(int n) {}
void A::a() {}

void *pa1 = __builtin_function_start((void(A::*)(int)) &A::a);
void *pa2 = __builtin_function_start((void(A::*)()) &A::a);
```

Description:

The `__builtin_function_start` builtin accepts an argument that can be constant-evaluated to a function, and returns the address of the function body. This builtin is not supported on all targets.

The returned pointer may differ from the normally taken function address and is not safe to call. For example, with `-fsanitize=cfi`, taking a function address produces a callable pointer to a CFI jump table, while `__builtin_function_start` returns an address that fails cfi-icall checks.

`__builtin_operator_new` and `__builtin_operator_delete`

A call to `__builtin_operator_new(args)` is exactly the same as a call to `::operator new(args)`, except that it allows certain optimizations that the C++ standard does not permit for a direct function call to `::operator new` (in particular, removing `new` / `delete` pairs and merging allocations), and that the call is required to resolve to a replaceable global allocation function.

Likewise, `__builtin_operator_delete` is exactly the same as a call to `::operator delete(args)`, except that it permits optimizations and that the call is required to resolve to a replaceable global deallocation function.

These builtins are intended for use in the implementation of `std::allocator` and other similar allocation libraries, and are only available in C++.

Query for this feature with `__has_builtin(__builtin_operator_new)` or `__has_builtin(__builtin_operator_delete)`:

- If the value is at least `201802L`, the builtins behave as described above.
- If the value is non-zero, the builtins may not support calling arbitrary replaceable global (de)allocation functions, but do support calling at least `::operator new(size_t)` and `::operator delete(void*)`.

`__builtin_preserve_access_index`

`__builtin_preserve_access_index` specifies a code section where array subscript access and structure/union member access are relocatable under bpf compile-once run-everywhere framework. Debuginfo (typically with `-g`) is needed, otherwise, the compiler will exit with an error. The return type for the intrinsic is the same as the type of the argument.

Syntax:

```
type __builtin_preserve_access_index(type arg)
```

Example of Use:

```
struct t {
  int i;
  int j;
  union {
    int a;
    int b;
  } c[4];
};
struct t *v = ...;
int *pb =__builtin_preserve_access_index(&v->c[3].b);
__builtin_preserve_access_index(v->j);
```

`__builtin_debugtrap`

`__builtin_debugtrap` causes the program to stop its execution in such a way that a debugger can catch it.

Syntax:

```
__builtin_debugtrap()
```

Description

`__builtin_debugtrap` is lowered to the `` `llvm.debugtrap <https://llvm.org/docs/LangRef.html#llvm-debugtrap-intrinsic>`_ `` builtin. It should have the same effect as setting a breakpoint on the line where the builtin is called.

Query for this feature with `__has_builtin(__builtin_debugtrap)`.

`__builtin_trap`

`__builtin_trap` causes the program to stop its execution abnormally.

Syntax:

```
__builtin_trap()
```

Description

`__builtin_trap` is lowered to the `` `llvm.trap <https://llvm.org/docs/LangRef.html#llvm-trap-intrinsic>`_ `` builtin.

Query for this feature with `__has_builtin(__builtin_trap)`.

`__builtin_sycl_unique_stable_name`

`__builtin_sycl_unique_stable_name()` is a builtin that takes a type and produces a string literal containing a unique name for the type that is stable across split compilations, mainly to support SYCL/Data Parallel C++ language.

In cases where the split compilation needs to share a unique token for a type across the boundary (such as in an offloading situation), this name can be used for lookup purposes, such as in the SYCL Integration Header.

The value of this builtin is computed entirely at compile time, so it can be used in constant expressions. This value encodes lambda functions based on a stable numbering order in which they appear in their local declaration contexts. Once this builtin is evaluated in a constexpr context, it is erroneous to use it in an instantiation which changes its value.

In order to produce the unique name, the current implementation of the bultin uses Itanium mangling even if the host compilation uses a different name mangling scheme at runtime. The mangler marks all the lambdas required to name the SYCL kernel and emits a stable local ordering of the respective lambdas. The resulting pattern is demanglable. When non-lambda types are passed to the builtin, the mangler emits their usual pattern without any special treatment.

Syntax:

```
// Computes a unique stable name for the given type.
constexpr const char * __builtin_sycl_unique_stable_name( type-id );
```

Multiprecision Arithmetic Builtins

Clang provides a set of builtins which expose multiprecision arithmetic in a manner amenable to C. They all have the following form:

```
unsigned x = ..., y = ..., carryin = ..., carryout;
unsigned sum = __builtin_addc(x, y, carryin, &carryout);
```

Thus one can form a multiprecision addition chain in the following manner:

```
unsigned *x, *y, *z, carryin=0, carryout;
z[0] = __builtin_addc(x[0], y[0], carryin, &carryout);
carryin = carryout;
z[1] = __builtin_addc(x[1], y[1], carryin, &carryout);
carryin = carryout;
z[2] = __builtin_addc(x[2], y[2], carryin, &carryout);
carryin = carryout;
z[3] = __builtin_addc(x[3], y[3], carryin, &carryout);
```

The complete list of builtins are:

```
unsigned char      __builtin_addcb (unsigned char x, unsigned char y, unsigned char carryin, unsigned char *carryout);
unsigned short     __builtin_addcs (unsigned short x, unsigned short y, unsigned short carryin, unsigned short *carryout);
unsigned           __builtin_addc  (unsigned x, unsigned y, unsigned carryin, unsigned *carryout);
unsigned long      __builtin_addcl (unsigned long x, unsigned long y, unsigned long carryin, unsigned long *carryout);
unsigned long long __builtin_addcll(unsigned long long x, unsigned long long y, unsigned long long carryin, unsigned long long *carryout);
unsigned char      __builtin_subcb (unsigned char x, unsigned char y, unsigned char carryin, unsigned char *carryout);
unsigned short     __builtin_subcs (unsigned short x, unsigned short y, unsigned short carryin, unsigned short *carryout);
unsigned           __builtin_subc  (unsigned x, unsigned y, unsigned carryin, unsigned *carryout);
unsigned long      __builtin_subcl (unsigned long x, unsigned long y, unsigned long carryin, unsigned long *carryout);
unsigned long long __builtin_subcll(unsigned long long x, unsigned long long y, unsigned long long carryin, unsigned long long *carryout);
```

Checked Arithmetic Builtins

Clang provides a set of builtins that implement checked arithmetic for security critical applications in a manner that is fast and easily expressible in C. As an example of their usage:

```
errorcode_t security_critical_application(...) {
  unsigned x, y, result;
  ...
  if (__builtin_mul_overflow(x, y, &result))
    return kErrorCodeHackers;
  ...
  use_multiply(result);
```

```
  ...
}
```

Clang provides the following checked arithmetic builtins:

```
bool __builtin_add_overflow   (type1 x, type2 y, type3 *sum);
bool __builtin_sub_overflow   (type1 x, type2 y, type3 *diff);
bool __builtin_mul_overflow   (type1 x, type2 y, type3 *prod);
bool __builtin_uadd_overflow  (unsigned x, unsigned y, unsigned *sum);
bool __builtin_uaddl_overflow (unsigned long x, unsigned long y, unsigned long *sum);
bool __builtin_uaddll_overflow(unsigned long long x, unsigned long long y, unsigned long long *sum);
bool __builtin_usub_overflow  (unsigned x, unsigned y, unsigned *diff);
bool __builtin_usubl_overflow (unsigned long x, unsigned long y, unsigned long *diff);
bool __builtin_usubll_overflow(unsigned long long x, unsigned long long y, unsigned long long *diff);
bool __builtin_umul_overflow  (unsigned x, unsigned y, unsigned *prod);
bool __builtin_umull_overflow (unsigned long x, unsigned long y, unsigned long *prod);
bool __builtin_umulll_overflow(unsigned long long x, unsigned long long y, unsigned long long *prod);
bool __builtin_sadd_overflow  (int x, int y, int *sum);
bool __builtin_saddl_overflow (long x, long y, long *sum);
bool __builtin_saddll_overflow(long long x, long long y, long long *sum);
bool __builtin_ssub_overflow  (int x, int y, int *diff);
bool __builtin_ssubl_overflow (long x, long y, long *diff);
bool __builtin_ssubll_overflow(long long x, long long y, long long *diff);
bool __builtin_smul_overflow  (int x, int y, int *prod);
bool __builtin_smull_overflow (long x, long y, long *prod);
bool __builtin_smulll_overflow(long long x, long long y, long long *prod);
```

Each builtin performs the specified mathematical operation on the first two arguments and stores the result in the third argument. If possible, the result will be equal to mathematically-correct result and the builtin will return 0. Otherwise, the builtin will return 1 and the result will be equal to the unique value that is equivalent to the mathematically-correct result modulo two raised to the *k* power, where *k* is the number of bits in the result type. The behavior of these builtins is well-defined for all argument values.

The first three builtins work generically for operands of any integer type, including boolean types. The operands need not have the same type as each other, or as the result. The other builtins may implicitly promote or convert their operands before performing the operation.

Query for this feature with `__has_builtin(__builtin_add_overflow)`, etc.

Floating point builtins

`__builtin_canonicalize`

```
double __builtin_canonicalize(double);
float __builtin_canonicalizef(float);
long double__builtin_canonicalizel(long double);
```

Returns the platform specific canonical encoding of a floating point number. This canonicalization is useful for implementing certain numeric primitives such as frexp. See LLVM canonicalize intrinsic for more information on the semantics.

String builtins

Clang provides constant expression evaluation support for builtins forms of the following functions from the C standard library headers `<string.h>` and `<wchar.h>`:

- `memchr`
- `memcmp` (and its deprecated BSD / POSIX alias `bcmp`)
- `strchr`
- `strcmp`
- `strlen`
- `strncmp`

- wcschr
- wcscmp
- wcslen
- wcsncmp
- wmemchr
- wmemcmp

In each case, the builtin form has the name of the C library function prefixed by __builtin_. Example:

```
void *p = __builtin_memchr("foobar", 'b', 5);
```

In addition to the above, one further builtin is provided:

```
char *__builtin_char_memchr(const char *haystack, int needle, size_t size);
```

__builtin_char_memchr(a, b, c) is identical to (char*)__builtin_memchr(a, b, c) except that its use is permitted within constant expressions in C++11 onwards (where a cast from void* to char* is disallowed in general).

Constant evaluation support for the __builtin_mem* functions is provided only for arrays of char, signed char, unsigned char, or char8_t, despite these functions accepting an argument of type const void*.

Support for constant expression evaluation for the above builtins can be detected with __has_feature(cxx_constexpr_string_builtins).

Memory builtins

Clang provides constant expression evaluation support for builtin forms of the following functions from the C standard library headers <string.h> and <wchar.h>:

- memcpy
- memmove
- wmemcpy
- wmemmove

In each case, the builtin form has the name of the C library function prefixed by __builtin_.

Constant evaluation support is only provided when the source and destination are pointers to arrays with the same trivially copyable element type, and the given size is an exact multiple of the element size that is no greater than the number of elements accessible through the source and destination operands.

Guaranteed inlined copy

```
void __builtin_memcpy_inline(void *dst, const void *src, size_t size);
```

__builtin_memcpy_inline has been designed as a building block for efficient memcpy implementations. It is identical to __builtin_memcpy but also guarantees not to call any external functions. See LLVM IR llvm.memcpy.inline intrinsic for more information.

This is useful to implement a custom version of memcpy, implement a libc memcpy or work around the absence of a libc.

Note that the *size* argument must be a compile time constant.

Note that this intrinsic cannot yet be called in a constexpr context.

Atomic Min/Max builtins with memory ordering

There are two atomic builtins with min/max in-memory comparison and swap. The syntax and semantics are similar to GCC-compatible __atomic_* builtins.

- __atomic_fetch_min

- `__atomic_fetch_max`

The builtins work with signed and unsigned integers and require to specify memory ordering. The return value is the original value that was stored in memory before comparison.

Example:

```
unsigned int val = __atomic_fetch_min(unsigned int *pi, unsigned int ui, __ATOMIC_RELAXED);
```

The third argument is one of the memory ordering specifiers `__ATOMIC_RELAXED`, `__ATOMIC_CONSUME`, `__ATOMIC_ACQUIRE`, `__ATOMIC_RELEASE`, `__ATOMIC_ACQ_REL`, or `__ATOMIC_SEQ_CST` following C++11 memory model semantics.

In terms or aquire-release ordering barriers these two operations are always considered as operations with *load-store* semantics, even when the original value is not actually modified after comparison.

__c11_atomic builtins

Clang provides a set of builtins which are intended to be used to implement C11's `<stdatomic.h>` header. These builtins provide the semantics of the `_explicit` form of the corresponding C11 operation, and are named with a `__c11_` prefix. The supported operations, and the differences from the corresponding C11 operations, are:

- `__c11_atomic_init`
- `__c11_atomic_thread_fence`
- `__c11_atomic_signal_fence`
- `__c11_atomic_is_lock_free` (The argument is the size of the `_Atomic(...)` object, instead of its address)
- `__c11_atomic_store`
- `__c11_atomic_load`
- `__c11_atomic_exchange`
- `__c11_atomic_compare_exchange_strong`
- `__c11_atomic_compare_exchange_weak`
- `__c11_atomic_fetch_add`
- `__c11_atomic_fetch_sub`
- `__c11_atomic_fetch_and`
- `__c11_atomic_fetch_or`
- `__c11_atomic_fetch_xor`
- `__c11_atomic_fetch_nand` (Nand is not presented in `<stdatomic.h>`)
- `__c11_atomic_fetch_max`
- `__c11_atomic_fetch_min`

The macros `__ATOMIC_RELAXED`, `__ATOMIC_CONSUME`, `__ATOMIC_ACQUIRE`, `__ATOMIC_RELEASE`, `__ATOMIC_ACQ_REL`, and `__ATOMIC_SEQ_CST` are provided, with values corresponding to the enumerators of C11's `memory_order` enumeration.

(Note that Clang additionally provides GCC-compatible `__atomic_*` builtins and OpenCL 2.0 `__opencl_atomic_*` builtins. The OpenCL 2.0 atomic builtins are an explicit form of the corresponding OpenCL 2.0 builtin function, and are named with a `__opencl_` prefix. The macros `__OPENCL_MEMORY_SCOPE_WORK_ITEM`, `__OPENCL_MEMORY_SCOPE_WORK_GROUP`, `__OPENCL_MEMORY_SCOPE_DEVICE`, `__OPENCL_MEMORY_SCOPE_ALL_SVM_DEVICES`, and `__OPENCL_MEMORY_SCOPE_SUB_GROUP` are provided, with values corresponding to the enumerators of OpenCL's `memory_scope` enumeration.)

Low-level ARM exclusive memory builtins

Clang provides overloaded builtins giving direct access to the three key ARM instructions for implementing atomic operations.

```
T __builtin_arm_ldrex(const volatile T *addr);
T __builtin_arm_ldaex(const volatile T *addr);
int __builtin_arm_strex(T val, volatile T *addr);
int __builtin_arm_stlex(T val, volatile T *addr);
void __builtin_arm_clrex(void);
```

The types T currently supported are:

- Integer types with width at most 64 bits (or 128 bits on AArch64).
- Floating-point types
- Pointer types.

Note that the compiler does not guarantee it will not insert stores which clear the exclusive monitor in between an `ldrex` type operation and its paired `strex`. In practice this is only usually a risk when the extra store is on the same cache line as the variable being modified and Clang will only insert stack stores on its own, so it is best not to use these operations on variables with automatic storage duration.

Also, loads and stores may be implicit in code written between the `ldrex` and `strex`. Clang will not necessarily mitigate the effects of these either, so care should be exercised.

For these reasons the higher level atomic primitives should be preferred where possible.

Non-temporal load/store builtins

Clang provides overloaded builtins allowing generation of non-temporal memory accesses.

```
T __builtin_nontemporal_load(T *addr);
void __builtin_nontemporal_store(T value, T *addr);
```

The types T currently supported are:

- Integer types.
- Floating-point types.
- Vector types.

Note that the compiler does not guarantee that non-temporal loads or stores will be used.

C++ Coroutines support builtins

Warning

This is a work in progress. Compatibility across Clang/LLVM releases is not guaranteed.

Clang provides experimental builtins to support C++ Coroutines as defined by https://wg21.link/P0057. The following four are intended to be used by the standard library to implement the `std::coroutine_handle` type.

Syntax:

```
void  __builtin_coro_resume(void *addr);
void  __builtin_coro_destroy(void *addr);
bool  __builtin_coro_done(void *addr);
void *__builtin_coro_promise(void *addr, int alignment, bool from_promise)
```

Example of use:

```
template <> struct coroutine_handle<void> {
  void resume() const { __builtin_coro_resume(ptr); }
  void destroy() const { __builtin_coro_destroy(ptr); }
  bool done() const { return __builtin_coro_done(ptr); }
  // ...
protected:
```

```
  void *ptr;
};

template <typename Promise> struct coroutine_handle : coroutine_handle<> {
  // ...
  Promise &promise() const {
    return *reinterpret_cast<Promise *>(
      __builtin_coro_promise(ptr, alignof(Promise), /*from-promise=*/false));
  }
  static coroutine_handle from_promise(Promise &promise) {
    coroutine_handle p;
    p.ptr = __builtin_coro_promise(&promise, alignof(Promise),
                                                    /*from-promise=*/true);
    return p;
  }
};
```

Other coroutine builtins are either for internal clang use or for use during development of the coroutine feature. See Coroutines in LLVM for more information on their semantics. Note that builtins matching the intrinsics that take token as the first parameter (llvm.coro.begin, llvm.coro.alloc, llvm.coro.free and llvm.coro.suspend) omit the token parameter and fill it to an appropriate value during the emission.

Syntax:

```
size_t __builtin_coro_size()
void  *__builtin_coro_frame()
void  *__builtin_coro_free(void *coro_frame)

void  *__builtin_coro_id(int align, void *promise, void *fnaddr, void *parts)
bool   __builtin_coro_alloc()
void  *__builtin_coro_begin(void *memory)
void   __builtin_coro_end(void *coro_frame, bool unwind)
char   __builtin_coro_suspend(bool final)
```

Note that there is no builtin matching the *llvm.coro.save* intrinsic. LLVM automatically will insert one if the first argument to *llvm.coro.suspend* is token *none*. If a user calls *__builin_suspend*, clang will insert *token none* as the first argument to the intrinsic.

Source location builtins

Clang provides builtins to support C++ standard library implementation of `std::source_location` as specified in C++20. With the exception of `__builtin_COLUMN`, these builtins are also implemented by GCC.

Syntax:

```
const char *__builtin_FILE();
const char *__builtin_FUNCTION();
unsigned    __builtin_LINE();
unsigned    __builtin_COLUMN(); // Clang only
const std::source_location::__impl *__builtin_source_location();
```

Example of use:

```
void my_assert(bool pred, int line = __builtin_LINE(), // Captures line of caller
               const char* file = __builtin_FILE(),
               const char* function = __builtin_FUNCTION()) {
  if (pred) return;
  printf("%s:%d assertion failed in function %s\n", file, line, function);
  std::abort();
}

struct MyAggregateType {
  int x;
```

```
  int line = __builtin_LINE(); // captures line where aggregate initialization occurs
};
static_assert(MyAggregateType{42}.line == __LINE__);

struct MyClassType {
  int line = __builtin_LINE(); // captures line of the constructor used during initialization
  constexpr MyClassType(int) { assert(line == __LINE__); }
};
```

Description:

The builtins `__builtin_LINE`, `__builtin_FUNCTION`, and `__builtin_FILE` return the values, at the "invocation point", for `__LINE__`, `__FUNCTION__`, and `__FILE__` respectively. `__builtin_COLUMN` similarly returns the column, though there is no corresponding macro. These builtins are constant expressions.

When the builtins appear as part of a default function argument the invocation point is the location of the caller. When the builtins appear as part of a default member initializer, the invocation point is the location of the constructor or aggregate initialization used to create the object. Otherwise the invocation point is the same as the location of the builtin.

When the invocation point of `__builtin_FUNCTION` is not a function scope the empty string is returned.

The builtin `__builtin_source_location` returns a pointer to constant static data of type `std::source_location::__impl`. This type must have already been defined, and must contain exactly four fields: `const char *_M_file_name`, `const char *_M_function_name`, `<any-integral-type> _M_line`, and `<any-integral-type> _M_column`. The fields will be populated in the same manner as the above four builtins, except that `_M_function_name` is populated with `__PRETTY_FUNCTION__` rather than `__FUNCTION__`.

Alignment builtins

Clang provides builtins to support checking and adjusting alignment of pointers and integers. These builtins can be used to avoid relying on implementation-defined behavior of arithmetic on integers derived from pointers. Additionally, these builtins retain type information and, unlike bitwise arithmetic, they can perform semantic checking on the alignment value.

Syntax:

```
Type __builtin_align_up(Type value, size_t alignment);
Type __builtin_align_down(Type value, size_t alignment);
bool __builtin_is_aligned(Type value, size_t alignment);
```

Example of use:

```
char* global_alloc_buffer;
void* my_aligned_allocator(size_t alloc_size, size_t alignment) {
  char* result = __builtin_align_up(global_alloc_buffer, alignment);
  // result now contains the value of global_alloc_buffer rounded up to the
  // next multiple of alignment.
  global_alloc_buffer = result + alloc_size;
  return result;
}

void* get_start_of_page(void* ptr) {
  return __builtin_align_down(ptr, PAGE_SIZE);
}

void example(char* buffer) {
   if (__builtin_is_aligned(buffer, 64)) {
     do_fast_aligned_copy(buffer);
   } else {
     do_unaligned_copy(buffer);
   }
}
```

```
// In addition to pointers, the builtins can also be used on integer types
// and are evaluatable inside constant expressions.
static_assert(__builtin_align_up(123, 64) == 128, "");
static_assert(__builtin_align_down(123u, 64) == 64u, "");
static_assert(!__builtin_is_aligned(123, 64), "");
```

Description:

The builtins `__builtin_align_up`, `__builtin_align_down`, return their first argument aligned up/down to the next multiple of the second argument. If the value is already sufficiently aligned, it is returned unchanged. The builtin `__builtin_is_aligned` returns whether the first argument is aligned to a multiple of the second argument. All of these builtins expect the alignment to be expressed as a number of bytes.

These builtins can be used for all integer types as well as (non-function) pointer types. For pointer types, these builtins operate in terms of the integer address of the pointer and return a new pointer of the same type (including qualifiers such as `const`) with an adjusted address. When aligning pointers up or down, the resulting value must be within the same underlying allocation or one past the end (see C17 6.5.6p8, C++ [expr.add]). This means that arbitrary integer values stored in pointer-type variables must not be passed to these builtins. For those use cases, the builtins can still be used, but the operation must be performed on the pointer cast to `uintptr_t`.

If Clang can determine that the alignment is not a power of two at compile time, it will result in a compilation failure. If the alignment argument is not a power of two at run time, the behavior of these builtins is undefined.

Non-standard C++11 Attributes

Clang's non-standard C++11 attributes live in the `clang` attribute namespace.

Clang supports GCC's `gnu` attribute namespace. All GCC attributes which are accepted with the `__attribute__((foo))` syntax are also accepted as `[[gnu::foo]]`. This only extends to attributes which are specified by GCC (see the list of GCC function attributes, GCC variable attributes, and GCC type attributes). As with the GCC implementation, these attributes must appertain to the *declarator-id* in a declaration, which means they must go either at the start of the declaration or immediately after the name being declared.

For example, this applies the GNU `unused` attribute to a and `f`, and also applies the GNU `noreturn` attribute to `f`.

```
[[gnu::unused]] int a, f [[gnu::noreturn]] ();
```

Target-Specific Extensions

Clang supports some language features conditionally on some targets.

ARM/AArch64 Language Extensions

Memory Barrier Intrinsics

Clang implements the `__dmb`, `__dsb` and `__isb` intrinsics as defined in the ARM C Language Extensions Release 2.0. Note that these intrinsics are implemented as motion barriers that block reordering of memory accesses and side effect instructions. Other instructions like simple arithmetic may be reordered around the intrinsic. If you expect to have no reordering at all, use inline assembly instead.

X86/X86-64 Language Extensions

The X86 backend has these language extensions:

Memory references to specified segments

Annotating a pointer with address space #256 causes it to be code generated relative to the X86 GS segment register, address space #257 causes it to be relative to the X86 FS segment, and address space #258 causes it to be relative to the X86 SS segment. Note that this is a very very low-level feature that should only be used if you know what you're doing (for example in an OS kernel).

Here is an example:

```
#define GS_RELATIVE __attribute__((address_space(256)))
int foo(int GS_RELATIVE *P) {
  return *P;
}
```

Which compiles to (on X86-32):

```
_foo:
        movl    4(%esp), %eax
        movl    %gs:(%eax), %eax
        ret
```

You can also use the GCC compatibility macros `__seg_fs` and `__seg_gs` for the same purpose. The preprocessor symbols `__SEG_FS` and `__SEG_GS` indicate their support.

PowerPC Language Extensions

Set the Floating Point Rounding Mode

PowerPC64/PowerPC64le supports the builtin function `__builtin_setrnd` to set the floating point rounding mode. This function will use the least significant two bits of integer argument to set the floating point rounding mode.

```
double __builtin_setrnd(int mode);
```

The effective values for mode are:

- 0 - round to nearest
- 1 - round to zero
- 2 - round to +infinity
- 3 - round to -infinity

Note that the mode argument will modulo 4, so if the integer argument is greater than 3, it will only use the least significant two bits of the mode. Namely, `__builtin_setrnd(102))` is equal to `__builtin_setrnd(2)`.

PowerPC cache builtins

The PowerPC architecture specifies instructions implementing cache operations. Clang provides builtins that give direct programmer access to these cache instructions.

Currently the following builtins are implemented in clang:

`__builtin_dcbf` copies the contents of a modified block from the data cache to main memory and flushes the copy from the data cache.

Syntax:

```
void __dcbf(const void* addr); /* Data Cache Block Flush */
```

Example of Use:

```
int a = 1;
__builtin_dcbf (&a);
```

Extensions for Static Analysis

Clang supports additional attributes that are useful for documenting program invariants and rules for static analysis tools, such as the Clang Static Analyzer. These attributes are documented in the analyzer's list of source-level annotations.

Extensions for Dynamic Analysis

Use `__has_feature(address_sanitizer)` to check if the code is being built with AddressSanitizer.

Use `__has_feature(thread_sanitizer)` to check if the code is being built with ThreadSanitizer.

Use `__has_feature(memory_sanitizer)` to check if the code is being built with MemorySanitizer.

Use `__has_feature(dataflow_sanitizer)` to check if the code is being built with DataFlowSanitizer.

Use `__has_feature(safe_stack)` to check if the code is being built with SafeStack.

Extensions for selectively disabling optimization

Clang provides a mechanism for selectively disabling optimizations in functions and methods.

To disable optimizations in a single function definition, the GNU-style or C++11 non-standard attribute `optnone` can be used.

```
// The following functions will not be optimized.
// GNU-style attribute
__attribute__((optnone)) int foo() {
  // ... code
}
// C++11 attribute
[[clang::optnone]] int bar() {
  // ... code
}
```

To facilitate disabling optimization for a range of function definitions, a range-based pragma is provided. Its syntax is `#pragma clang optimize` followed by `off` or `on`.

All function definitions in the region between an `off` and the following `on` will be decorated with the `optnone` attribute unless doing so would conflict with explicit attributes already present on the function (e.g. the ones that control inlining).

```
#pragma clang optimize off
// This function will be decorated with optnone.
int foo() {
  // ... code
}

// optnone conflicts with always_inline, so bar() will not be decorated.
__attribute__((always_inline)) int bar() {
  // ... code
}
#pragma clang optimize on
```

If no `on` is found to close an `off` region, the end of the region is the end of the compilation unit.

Note that a stray `#pragma clang optimize on` does not selectively enable additional optimizations when compiling at low optimization levels. This feature can only be used to selectively disable optimizations.

The pragma has an effect on functions only at the point of their definition; for function templates, this means that the state of the pragma at the point of an instantiation is not necessarily relevant. Consider the following example:

```
template<typename T> T twice(T t) {
  return 2 * t;
}

#pragma clang optimize off
template<typename T> T thrice(T t) {
  return 3 * t;
}

int container(int a, int b) {
  return twice(a) + thrice(b);
}
#pragma clang optimize on
```

In this example, the definition of the template function `twice` is outside the pragma region, whereas the definition of `thrice` is inside the region. The `container` function is also in the region and will not be optimized, but it causes the instantiation of `twice` and `thrice` with an `int` type; of these two instantiations, `twice` will be optimized (because its definition was outside the region) and `thrice` will not be optimized.

Extensions for loop hint optimizations

The `#pragma clang loop` directive is used to specify hints for optimizing the subsequent for, while, do-while, or c++11 range-based for loop. The directive provides options for vectorization, interleaving, predication, unrolling and distribution. Loop hints can be specified before any loop and will be ignored if the optimization is not safe to apply.

There are loop hints that control transformations (e.g. vectorization, loop unrolling) and there are loop hints that set transformation options (e.g. `vectorize_width`, `unroll_count`). Pragmas setting transformation options imply the transformation is enabled, as if it was enabled via the corresponding transformation pragma (e.g. `vectorize(enable)`). If the transformation is disabled (e.g. `vectorize(disable)`), that takes precedence over transformations option pragmas implying that transformation.

Vectorization, Interleaving, and Predication

A vectorized loop performs multiple iterations of the original loop in parallel using vector instructions. The instruction set of the target processor determines which vector instructions are available and their vector widths. This restricts the types of loops that can be vectorized. The vectorizer automatically determines if the loop is safe and profitable to vectorize. A vector instruction cost model is used to select the vector width.

Interleaving multiple loop iterations allows modern processors to further improve instruction-level parallelism (ILP) using advanced hardware features, such as multiple execution units and out-of-order execution. The vectorizer uses a cost model that depends on the register pressure and generated code size to select the interleaving count.

Vectorization is enabled by `vectorize(enable)` and interleaving is enabled by `interleave(enable)`. This is useful when compiling with `-Os` to manually enable vectorization or interleaving.

```
#pragma clang loop vectorize(enable)
#pragma clang loop interleave(enable)
for(...) {
  ...
}
```

The vector width is specified by `vectorize_width(_value_[, fixed|scalable])`, where _value_ is a positive integer and the type of vectorization can be specified with an optional second parameter. The default for the second parameter is 'fixed' and refers to fixed width vectorization, whereas 'scalable' indicates the compiler should use scalable vectors instead. Another use of vectorize_width is `vectorize_width(fixed|scalable)` where the user can hint at the type of vectorization to use without specifying the exact width. In both variants of the pragma the vectorizer may decide to fall back on fixed width vectorization if the target does not support scalable vectors.

The interleave count is specified by `interleave_count(_value_)`, where _value_ is a positive integer. This is useful for specifying the optimal width/count of the set of target architectures supported by your application.

```
#pragma clang loop vectorize_width(2)
#pragma clang loop interleave_count(2)
for(...) {
  ...
}
```

Specifying a width/count of `1` disables the optimization, and is equivalent to `vectorize(disable)` or `interleave(disable)`.

Vector predication is enabled by `vectorize_predicate(enable)`, for example:

```
#pragma clang loop vectorize(enable)
#pragma clang loop vectorize_predicate(enable)
for(...) {
  ...
}
```

This predicates (masks) all instructions in the loop, which allows the scalar remainder loop (the tail) to be folded into the main vectorized loop. This might be more efficient when vector predication is efficiently supported by the target platform.

Loop Unrolling

Unrolling a loop reduces the loop control overhead and exposes more opportunities for ILP. Loops can be fully or partially unrolled. Full unrolling eliminates the loop and replaces it with an enumerated sequence of loop iterations. Full unrolling is only possible if the loop trip count is known at compile time. Partial unrolling replicates the loop body within the loop and reduces the trip count.

If `unroll(enable)` is specified the unroller will attempt to fully unroll the loop if the trip count is known at compile time. If the fully unrolled code size is greater than an internal limit the loop will be partially unrolled up to this limit. If the trip count is not known at compile time the loop will be partially unrolled with a heuristically chosen unroll factor.

```
#pragma clang loop unroll(enable)
for(...) {
  ...
}
```

If `unroll(full)` is specified the unroller will attempt to fully unroll the loop if the trip count is known at compile time identically to `unroll(enable)`. However, with `unroll(full)` the loop will not be unrolled if the loop count is not known at compile time.

```
#pragma clang loop unroll(full)
for(...) {
  ...
}
```

The unroll count can be specified explicitly with `unroll_count(_value_)` where _value_ is a positive integer. If this value is greater than the trip count the loop will be fully unrolled. Otherwise the loop is partially unrolled subject to the same code size limit as with `unroll(enable)`.

```
#pragma clang loop unroll_count(8)
for(...) {
  ...
}
```

Unrolling of a loop can be prevented by specifying `unroll(disable)`.

Loop unroll parameters can be controlled by options *-mllvm -unroll-count=n* and *-mllvm -pragma-unroll-threshold=n*.

Loop Distribution

Loop Distribution allows splitting a loop into multiple loops. This is beneficial for example when the entire loop cannot be vectorized but some of the resulting loops can.

If `distribute(enable))` is specified and the loop has memory dependencies that inhibit vectorization, the compiler will attempt to isolate the offending operations into a new loop. This optimization is not enabled by default, only loops marked with the pragma are considered.

```
#pragma clang loop distribute(enable)
for (i = 0; i < N; ++i) {
  S1: A[i + 1] = A[i] + B[i];
  S2: C[i] = D[i] * E[i];
}
```

This loop will be split into two loops between statements S1 and S2. The second loop containing S2 will be vectorized.

Loop Distribution is currently not enabled by default in the optimizer because it can hurt performance in some cases. For example, instruction-level parallelism could be reduced by sequentializing the execution of the statements S1 and S2 above.

If Loop Distribution is turned on globally with `-mllvm -enable-loop-distribution`, specifying `distribute(disable)` can be used the disable it on a per-loop basis.

Additional Information

For convenience multiple loop hints can be specified on a single line.

```
#pragma clang loop vectorize_width(4) interleave_count(8)
for(...) {
  ...
}
```

If an optimization cannot be applied any hints that apply to it will be ignored. For example, the hint `vectorize_width(4)` is ignored if the loop is not proven safe to vectorize. To identify and diagnose optimization issues use *-Rpass*, *-Rpass-missed*, and *-Rpass-analysis* command line options. See the user guide for details.

Extensions to specify floating-point flags

The `#pragma clang fp` pragma allows floating-point options to be specified for a section of the source code. This pragma can only appear at file scope or at the start of a compound statement (excluding comments). When using within a compound statement, the pragma is active within the scope of the compound statement.

Currently, the following settings can be controlled with this pragma:

`#pragma clang fp reassociate` allows control over the reassociation of floating point expressions. When enabled, this pragma allows the expression `x + (y + z)` to be reassociated as `(x + y) + z`. Reassociation can also occur across multiple statements. This pragma can be used to disable reassociation when it is otherwise enabled for the translation unit with the `-fassociative-math` flag. The pragma can take two values: `on` and `off`.

```
float f(float x, float y, float z)
{
  // Enable floating point reassociation across statements
  #pragma clang fp reassociate(on)
  float t = x + y;
  float v = t + z;
}
```

`#pragma clang fp contract` specifies whether the compiler should contract a multiply and an addition (or subtraction) into a fused FMA operation when supported by the target.

The pragma can take three values: `on`, `fast` and `off`. The `on` option is identical to using `#pragma STDC FP_CONTRACT(ON)` and it allows fusion as specified the language standard. The `fast` option allows fusion in cases when the language standard does not make this possible (e.g. across statements in C).

```
for(...) {
  #pragma clang fp contract(fast)
  a = b[i] * c[i];
  d[i] += a;
}
```

The pragma can also be used with `off` which turns FP contraction off for a section of the code. This can be useful when fast contraction is otherwise enabled for the translation unit with the `-ffp-contract=fast-honor-pragmas` flag. Note that `-ffp-contract=fast` will override pragmas to fuse multiply and addition across statements regardless of any controlling pragmas.

`#pragma clang fp exceptions` specifies floating point exception behavior. It may take one of the values: `ignore`, `maytrap` or `strict`. Meaning of these values is same as for constrained floating point intrinsics.

```
{
  // Preserve floating point exceptions
  #pragma clang fp exceptions(strict)
  z = x + y;
  if (fetestexcept(FE_OVERFLOW))
        ...
}
```

A `#pragma clang fp` pragma may contain any number of options:

```
void func(float *dest, float a, float b) {
  #pragma clang fp exceptions(maytrap) contract(fast) reassociate(on)
  ...
}
```

`#pragma clang fp eval_method` allows floating-point behavior to be specified for a section of the source code. This pragma can appear at file or namespace scope, or at the start of a compound statement (excluding comments). The pragma is active within the scope of the compound statement.

When `pragma clang fp eval_method(source)` is enabled, the section of code governed by the pragma behaves as though the command-line option `-ffp-eval-method=source` is enabled. Rounds intermediate results to source-defined precision.

When `pragma clang fp eval_method(double)` is enabled, the section of code governed by the pragma behaves as though the command-line option `-ffp-eval-method=double` is enabled. Rounds intermediate results to `double` precision.

When `pragma clang fp eval_method(extended)` is enabled, the section of code governed by the pragma behaves as though the command-line option `-ffp-eval-method=extended` is enabled. Rounds intermediate results to target-dependent `long double` precision. In Win32 programming, for instance, the long double data type maps to the double, 64-bit precision data type.

The full syntax this pragma supports is `#pragma clang fp eval_method(source|double|extended)`.

```
for(...) {
  // The compiler will use long double as the floating-point evaluation
  // method.
  #pragma clang fp eval_method(extended)
  a = b[i] * c[i] + e;
}
```

The `#pragma float_control` pragma allows precise floating-point semantics and floating-point exception behavior to be specified for a section of the source code. This pragma can only appear at file or namespace scope, within a language linkage specification or at the start of a compound statement (excluding comments). When used within a compound statement, the pragma is active within the scope of the compound statement. This pragma is modeled after a Microsoft pragma with the same spelling and syntax. For pragmas specified at file or namespace scope, or within a language linkage specification, a stack is supported so that the `pragma float_control` settings can be pushed or popped.

When `pragma float_control(precise, on)` is enabled, the section of code governed by the pragma uses precise floating point semantics, effectively `-ffast-math` is disabled and `-ffp-contract=on` (fused multiply add) is enabled.

When `pragma float_control(except, on)` is enabled, the section of code governed by the pragma behaves as though the command-line option `-ffp-exception-behavior=strict` is enabled, when `pragma float_control(except, off)` is enabled, the section of code governed by the pragma behaves as though the command-line option `-ffp-exception-behavior=ignore` is enabled.

The full syntax this pragma supports is `float_control(except|precise, on|off [, push])` and `float_control(push|pop)`. The `push` and `pop` forms, including using `push` as the optional third argument, can only occur at file scope.

```
for(...) {
  // This block will be compiled with -fno-fast-math and -ffp-contract=on
  #pragma float_control(precise, on)
  a = b[i] * c[i] + e;
}
```

Specifying an attribute for multiple declarations (#pragma clang attribute)

The `#pragma clang attribute` directive can be used to apply an attribute to multiple declarations. The `#pragma clang attribute push` variation of the directive pushes a new "scope" of `#pragma clang attribute` that attributes can be added to. The `#pragma clang attribute (...)` variation adds an attribute to that scope, and the `#pragma clang attribute pop` variation pops the scope. You can also use `#pragma clang attribute push (...)`, which is a shorthand for when you want to add one attribute to a new scope. Multiple push directives can be nested inside each other.

The attributes that are used in the `#pragma clang attribute` directives can be written using the GNU-style syntax:

```
#pragma clang attribute push (__attribute__((annotate("custom"))), apply_to = function)

void function(); // The function now has the annotate("custom") attribute

#pragma clang attribute pop
```

The attributes can also be written using the C++11 style syntax:

```
#pragma clang attribute push ([[noreturn]], apply_to = function)

void function(); // The function now has the [[noreturn]] attribute

#pragma clang attribute pop
```

The `__declspec` style syntax is also supported:

```
#pragma clang attribute push (__declspec(dllexport), apply_to = function)

void function(); // The function now has the __declspec(dllexport) attribute

#pragma clang attribute pop
```

A single push directive can contain multiple attributes, however, only one syntax style can be used within a single directive:

```
#pragma clang attribute push ([[noreturn, noinline]], apply_to = function)

void function1(); // The function now has the [[noreturn]] and [[noinline]] attributes

#pragma clang attribute pop

#pragma clang attribute push (__attribute((noreturn, noinline)), apply_to = function)

void function2(); // The function now has the __attribute((noreturn)) and __attribute((noinline)) attributes

#pragma clang attribute pop
```

Because multiple push directives can be nested, if you're writing a macro that expands to `_Pragma("clang attribute")` it's good hygiene (though not required) to add a namespace to your push/pop directives. A pop directive with a namespace will pop the innermost push that has that same namespace. This will ensure that another macro's `pop` won't inadvertently pop your attribute. Note that an `pop` without a namespace will pop the innermost `push` without a namespace. `push``es with a namespace can only be popped by ``pop` with the same namespace. For instance:

```
#define ASSUME_NORETURN_BEGIN _Pragma("clang attribute AssumeNoreturn.push ([[noreturn]], apply_to = function)")
#define ASSUME_NORETURN_END   _Pragma("clang attribute AssumeNoreturn.pop")

#define ASSUME_UNAVAILABLE_BEGIN _Pragma("clang attribute Unavailable.push (__attribute__((unavailable)), apply_to=function)")
#define ASSUME_UNAVAILABLE_END   _Pragma("clang attribute Unavailable.pop")


ASSUME_NORETURN_BEGIN
ASSUME_UNAVAILABLE_BEGIN
void function(); // function has [[noreturn]] and __attribute__((unavailable))
ASSUME_NORETURN_END
void other_function(); // function has __attribute__((unavailable))
ASSUME_UNAVAILABLE_END
```

Without the namespaces on the macros, `other_function` will be annotated with `[[noreturn]]` instead of `__attribute__((unavailable))`. This may seem like a contrived example, but its very possible for this kind of

situation to appear in real code if the pragmas are spread out across a large file. You can test if your version of clang supports namespaces on `#pragma clang attribute` with `__has_extension(pragma_clang_attribute_namespaces)`.

Subject Match Rules

The set of declarations that receive a single attribute from the attribute stack depends on the subject match rules that were specified in the pragma. Subject match rules are specified after the attribute. The compiler expects an identifier that corresponds to the subject set specifier. The `apply_to` specifier is currently the only supported subject set specifier. It allows you to specify match rules that form a subset of the attribute's allowed subject set, i.e. the compiler doesn't require all of the attribute's subjects. For example, an attribute like `[[nodiscard]]` whose subject set includes `enum`, `record` and `hasType(functionType)`, requires the presence of at least one of these rules after `apply_to`:

```
#pragma clang attribute push([[nodiscard]], apply_to = enum)

enum Enum1 { A1, B1 }; // The enum will receive [[nodiscard]]

struct Record1 { }; // The struct will *not* receive [[nodiscard]]

#pragma clang attribute pop

#pragma clang attribute push([[nodiscard]], apply_to = any(record, enum))

enum Enum2 { A2, B2 }; // The enum will receive [[nodiscard]]

struct Record2 { }; // The struct *will* receive [[nodiscard]]

#pragma clang attribute pop

// This is an error, since [[nodiscard]] can't be applied to namespaces:
#pragma clang attribute push([[nodiscard]], apply_to = any(record, namespace))

#pragma clang attribute pop
```

Multiple match rules can be specified using the `any` match rule, as shown in the example above. The `any` rule applies attributes to all declarations that are matched by at least one of the rules in the `any`. It doesn't nest and can't be used inside the other match rules. Redundant match rules or rules that conflict with one another should not be used inside of `any`. Failing to specify a rule within the `any` rule results in an error.

Clang supports the following match rules:

- `function`: Can be used to apply attributes to functions. This includes C++ member functions, static functions, operators, and constructors/destructors.
- `function(is_member)`: Can be used to apply attributes to C++ member functions. This includes members like static functions, operators, and constructors/destructors.
- `hasType(functionType)`: Can be used to apply attributes to functions, C++ member functions, and variables/fields whose type is a function pointer. It does not apply attributes to Objective-C methods or blocks.
- `type_alias`: Can be used to apply attributes to `typedef` declarations and C++11 type aliases.
- `record`: Can be used to apply attributes to `struct`, `class`, and `union` declarations.
- `record(unless(is_union))`: Can be used to apply attributes only to `struct` and `class` declarations.
- `enum`: Can be be used to apply attributes to enumeration declarations.
- `enum_constant`: Can be used to apply attributes to enumerators.
- `variable`: Can be used to apply attributes to variables, including local variables, parameters, global variables, and static member variables. It does not apply attributes to instance member variables or Objective-C ivars.
- `variable(is_thread_local)`: Can be used to apply attributes to thread-local variables only.
- `variable(is_global)`: Can be used to apply attributes to global variables only.

- `variable(is_local)`: Can be used to apply attributes to local variables only.
- `variable(is_parameter)`: Can be used to apply attributes to parameters only.
- `variable(unless(is_parameter))`: Can be used to apply attributes to all the variables that are not parameters.
- `field`: Can be used to apply attributes to non-static member variables in a record. This includes Objective-C ivars.
- `namespace`: Can be used to apply attributes to `namespace` declarations.
- `objc_interface`: Can be used to apply attributes to `@interface` declarations.
- `objc_protocol`: Can be used to apply attributes to `@protocol` declarations.
- `objc_category`: Can be used to apply attributes to category declarations, including class extensions.
- `objc_method`: Can be used to apply attributes to Objective-C methods, including instance and class methods. Implicit methods like implicit property getters and setters do not receive the attribute.
- `objc_method(is_instance)`: Can be used to apply attributes to Objective-C instance methods.
- `objc_property`: Can be used to apply attributes to `@property` declarations.
- `block`: Can be used to apply attributes to block declarations. This does not include variables/fields of block pointer type.

The use of `unless` in match rules is currently restricted to a strict set of sub-rules that are used by the supported attributes. That means that even though `variable(unless(is_parameter))` is a valid match rule, `variable(unless(is_thread_local))` is not.

Supported Attributes

Not all attributes can be used with the `#pragma clang attribute` directive. Notably, statement attributes like `[[fallthrough]]` or type attributes like `address_space` aren't supported by this directive. You can determine whether or not an attribute is supported by the pragma by referring to the individual documentation for that attribute.

The attributes are applied to all matching declarations individually, even when the attribute is semantically incorrect. The attributes that aren't applied to any declaration are not verified semantically.

Specifying section names for global objects (#pragma clang section)

The `#pragma clang section` directive provides a means to assign section-names to global variables, functions and static variables.

The section names can be specified as:

```
#pragma clang section bss="myBSS" data="myData" rodata="myRodata" relro="myRelro" text="myText"
```

The section names can be reverted back to default name by supplying an empty string to the section kind, for example:

```
#pragma clang section bss="" data="" text="" rodata="" relro=""
```

The `#pragma clang section` directive obeys the following rules:

- The pragma applies to all global variable, statics and function declarations from the pragma to the end of the translation unit.
- The pragma clang section is enabled automatically, without need of any flags.
- This feature is only defined to work sensibly for ELF targets.
- If section name is specified through _attribute_((section("myname"))), then the attribute name gains precedence.
- Global variables that are initialized to zero will be placed in the named bss section, if one is present.

- The `#pragma clang section` directive does not does try to infer section-kind from the name. For example, naming a section "`.bss.mySec`" does NOT mean it will be a bss section name.
- The decision about which section-kind applies to each global is taken in the back-end. Once the section-kind is known, appropriate section name, as specified by the user using `#pragma clang section` directive, is applied to that global.

Specifying Linker Options on ELF Targets

The `#pragma comment(lib, ...)` directive is supported on all ELF targets. The second parameter is the library name (without the traditional Unix prefix of `lib`). This allows you to provide an implicit link of dependent libraries.

Evaluating Object Size Dynamically

Clang supports the builtin `__builtin_dynamic_object_size`, the semantics are the same as GCC's `__builtin_object_size` (which Clang also supports), but `__builtin_dynamic_object_size` can evaluate the object's size at runtime. `__builtin_dynamic_object_size` is meant to be used as a drop-in replacement for `__builtin_object_size` in libraries that support it.

For instance, here is a program that `__builtin_dynamic_object_size` will make safer:

```
void copy_into_buffer(size_t size) {
  char* buffer = malloc(size);
  strlcpy(buffer, "some string", strlen("some string"));
  // Previous line preprocesses to:
  // __builtin___strlcpy_chk(buffer, "some string", strlen("some string"), __builtin_object_size(buffer, 0))
}
```

Since the size of `buffer` can't be known at compile time, Clang will fold `__builtin_object_size(buffer, 0)` into `-1`. However, if this was written as `__builtin_dynamic_object_size(buffer, 0)`, Clang will fold it into `size`, providing some extra runtime safety.

Deprecating Macros

Clang supports the pragma `#pragma clang deprecated`, which can be used to provide deprecation warnings for macro uses. For example:

```
#define MIN(x, y) x < y ? x : y
#pragma clang deprecated(MIN, "use std::min instead")

void min(int a, int b) {
  return MIN(a, b); // warning: MIN is deprecated: use std::min instead
}
```

`#pragma clang deprecated` should be preferred for this purpose over `#pragma GCC warning` because the warning can be controlled with `-Wdeprecated`.

Restricted Expansion Macros

Clang supports the pragma `#pragma clang restrict_expansion`, which can be used restrict macro expansion in headers. This can be valuable when providing headers with ABI stability requirements. Any expansion of the annotated macro processed by the preprocessor after the `#pragma` annotation will log a warning. Redefining the macro or undefining the macro will not be diagnosed, nor will expansion of the macro within the main source file. For example:

```
#define TARGET_ARM 1
#pragma clang restrict_expansion(TARGET_ARM, "<reason>")

/// Foo.h
struct Foo {
#if TARGET_ARM // warning: TARGET_ARM is marked unsafe in headers: <reason>
  uint32_t X;
#else
  uint64_t X;
```

```
#endif
};

/// main.c
#include "foo.h"
#if TARGET_ARM // No warning in main source file
X_TYPE uint32_t
#else
X_TYPE uint64_t
#endif
```

This warning is controlled by `-Wpedantic-macros`.

Final Macros

Clang supports the pragma `#pragma clang final`, which can be used to mark macros as final, meaning they cannot be undef'd or re-defined. For example:

```
#define FINAL_MACRO 1
#pragma clang final(FINAL_MACRO)

#define FINAL_MACRO // warning: FINAL_MACRO is marked final and should not be redefined
#undef FINAL_MACRO  // warning: FINAL_MACRO is marked final and should not be undefined
```

This is useful for enforcing system-provided macros that should not be altered in user headers or code. This is controlled by `-Wpedantic-macros`. Final macros will always warn on redefinition, including situations with identical bodies and in system headers.

Line Control

Clang supports an extension for source line control, which takes the form of a preprocessor directive starting with an unsigned integral constant. In addition to the standard `#line` directive, this form allows control of an include stack and header file type, which is used in issuing diagnostics. These lines are emitted in preprocessed output.

```
# <line:number> <filename:string> <header-type:numbers>
```

The filename is optional, and if unspecified indicates no change in source filename. The header-type is an optional, whitespace-delimited, sequence of magic numbers as follows.

- `1:` Push the current source file name onto the include stack and enter a new file.
- 2: Pop the include stack and return to the specified file. If the filename is `""`, the name popped from the include stack is used. Otherwise there is no requirement that the specified filename matches the current source when originally pushed.
- 3: Enter a system-header region. System headers often contain implementation-specific source that would normally emit a diagnostic.
- 4: Enter an implicit `extern "C"` region. This is not required on modern systems where system headers are C++-aware.

At most a single 1 or 2 can be present, and values must be in ascending order.

Examples are:

```
# 57 // Advance (or return) to line 57 of the current source file
# 57 "frob" // Set to line 57 of "frob"
# 1 "foo.h" 1 // Enter "foo.h" at line 1
# 59 "main.c" 2 // Leave current include and return to "main.c"
# 1 "/usr/include/stdio.h" 1 3 // Enter a system header
# 60 "" 2 // return to "main.c"
# 1 "/usr/ancient/header.h" 1 4 // Enter an implicit extern "C" header
```

Extended Integer Types

Clang supports the C23 `_BitInt(N)` feature as an extension in older C modes and in C++. This type was previously implemented in Clang with the same semantics, but spelled `_ExtInt(N)`. This spelling has been deprecated in favor of the standard type.

Note: the ABI for `_BitInt(N)` is still in the process of being stabilized, so this type should not yet be used in interfaces that require ABI stability.

Intrinsics Support within Constant Expressions

The following builtin intrinsics can be used in constant expressions:

- `__builtin_bitreverse8`
- `__builtin_bitreverse16`
- `__builtin_bitreverse32`
- `__builtin_bitreverse64`
- `__builtin_bswap16`
- `__builtin_bswap32`
- `__builtin_bswap64`
- `__builtin_clrsb`
- `__builtin_clrsbl`
- `__builtin_clrsbll`
- `__builtin_clz`
- `__builtin_clzl`
- `__builtin_clzll`
- `__builtin_clzs`
- `__builtin_ctz`
- `__builtin_ctzl`
- `__builtin_ctzll`
- `__builtin_ctzs`
- `__builtin_ffs`
- `__builtin_ffsl`
- `__builtin_ffsll`
- `__builtin_fpclassify`
- `__builtin_inf`
- `__builtin_isinf`
- `__builtin_isinf_sign`
- `__builtin_isfinite`
- `__builtin_isnan`
- `__builtin_isnormal`
- `__builtin_nan`
- `__builtin_nans`
- `__builtin_parity`
- `__builtin_parityl`

- `__builtin_parityll`
- `__builtin_popcount`
- `__builtin_popcountl`
- `__builtin_popcountll`
- `__builtin_rotateleft8`
- `__builtin_rotateleft16`
- `__builtin_rotateleft32`
- `__builtin_rotateleft64`
- `__builtin_rotateright8`
- `__builtin_rotateright16`
- `__builtin_rotateright32`
- `__builtin_rotateright64`

The following x86-specific intrinsics can be used in constant expressions:

- `_bit_scan_forward`
- `_bit_scan_reverse`
- `__bsfd`
- `__bsfq`
- `__bsrd`
- `__bsrq`
- `__bswap`
- `__bswapd`
- `__bswap64`
- `__bswapq`
- `_castf32_u32`
- `_castf64_u64`
- `_castu32_f32`
- `_castu64_f64`
- `_mm_popcnt_u32`
- `_mm_popcnt_u64`
- `_popcnt32`
- `_popcnt64`
- `__popcntd`
- `__popcntq`
- `__rolb`
- `__rolw`
- `__rold`
- `__rolq`
- `__rorb`
- `__rorw`
- `__rord`
- `__rorq`

- `_rotl`
- `_rotr`
- `_rotwl`
- `_rotwr`
- `_lrotl`
- `_lrotr`

Clang command line argument reference

Introduction 184
Actions 193
Compilation flags 194
Preprocessor flags 197
Include path management 198
Dependency file generation 200
Dumping preprocessor state 200
Diagnostic flags 201
Target-independent compilation options 201
OpenCL flags 219
SYCL flags 220
Target-dependent compilation options 220
AARCH64 226
AMDGPU 227
ARM 227
Hexagon 228
Hexagon 228
M68k 228
MIPS 229
PowerPC 230
WebAssembly 231
WebAssembly Driver 231
X86 231
RISCV 233
Long double flags 234
Optimization level 234
Debug information generation 234
Kind and level of debug information 234
Debug level 234
Debugger to tune debug information for 235
Debug information flags 235
Static analyzer flags 235
Fortran compilation flags 235
Linker flags 237
<clang-dxc options> 238

Introduction

This page lists the command line arguments currently supported by the GCC-compatible `clang` and `clang++` drivers.

-B<prefix>, **--prefix** <arg>, **--prefix**=<arg>

Search $prefix$file for executables, libraries, and data files. If $prefix is a directory, search $prefix/$file

-F<arg>

Add directory to framework include search path

-ObjC

Treat source input files as Objective-C inputs

-ObjC++

Treat source input files as Objective-C++ inputs

-Qn, **-fno-ident**

Do not emit metadata containing compiler name and version

-Qunused-arguments

Don't emit warning for unused driver arguments

-Qy, **-fident**

Emit metadata containing compiler name and version

-Wa,<arg>,<arg2>...

Pass the comma separated arguments in <arg> to the assembler

-Wlarge-by-value-copy=<arg>

-Xarch_<arg1> <arg2>

-Xarch_device <arg>

Pass <arg> to the CUDA/HIP device compilation

-Xarch_host <arg>

Pass <arg> to the CUDA/HIP host compilation

-Xcuda-fatbinary <arg>

Pass <arg> to fatbinary invocation

-Xcuda-ptxas <arg>

Pass <arg> to the ptxas assembler

-Z<arg>

-a<arg>, **--profile-blocks**

-all_load

-allowable_client <arg>

--analyze

Run the static analyzer

--analyzer-no-default-checks

--analyzer-output<arg>

Static analyzer report output format (html|plist|plist-multi-file|plist-html|sarif|sarif-html|text).

-arch <arg>

-arch_errors_fatal

-arch_only <arg>

-arcmt-migrate-emit-errors

Emit ARC errors even if the migrator can fix them

-arcmt-migrate-report-output <arg>

Output path for the plist report

--autocomplete=<arg>

-bind_at_load

-bundle

-bundle_loader <arg>

-client_name<arg>

-compatibility_version<arg>

--config <arg>

Specifies configuration file

--constant-cfstrings

--cuda-feature=<arg>

Manually specify the CUDA feature to use

--cuda-include-ptx=<arg>, **--no-cuda-include-ptx**=<arg>

Include PTX for the following GPU architecture (e.g. sm_35) or ‘all’. May be specified more than once.

--cuda-noopt-device-debug, **--no-cuda-noopt-device-debug**

Enable device-side debug info generation. Disables ptxas optimizations.

-cuid=<arg>

An ID for compilation unit, which should be the same for the same compilation unit but different for different compilation units. It is used to externalize device-side static variables for single source offloading languages CUDA and HIP so that they can be accessed by the host code of the same compilation unit.

-current_version<arg>

-darwin-target-variant <arg>

Generate code for an additional runtime variant of the deployment target

-darwin-target-variant-triple <arg>

Specify the darwin target variant triple

-dead_strip

-dependency-dot <arg>

Filename to write DOT-formatted header dependencies to

-dependency-file <arg>

Filename (or -) to write dependency output to

-dsym-dir<dir>

Directory to output dSYM’s (if any) to

-dumpmachine

-dumpversion

--dyld-prefix=<arg>, **--dyld-prefix** <arg>

-dylib_file <arg>

-dylinker

-dylinker_install_name<arg>

-dynamic

-dynamiclib

-emit-ast

Emit Clang AST files for source inputs

--emit-static-lib

Enable linker job to emit a static library.

-enable-trivial-auto-var-init-zero-knowing-it-will-be-removed-from-clang

Trivial automatic variable initialization to zero is only here for benchmarks, it'll eventually be removed, and I'm OK with that because I'm only using it to benchmark

--end-no-unused-arguments

Start emitting warnings for unused driver arguments

-exported_symbols_list <arg>

-faligned-new=<arg>

-fautomatic

-ffat-lto-objects, **-fno-fat-lto-objects**

Embed the bitcode into the module and generate object code from an -flto compile.

-ffixed-r19

Reserve register r19 (Hexagon only)

-fgpu-default-stream=<arg>

Specify default stream. The default value is 'legacy'. (HIP only). <arg> must be 'legacy' or 'per-thread'.

-fgpu-flush-denormals-to-zero, **-fcuda-flush-denormals-to-zero**, **-fno-gpu-flush-denormals-to-zero**

Flush denormal floating point values to zero in CUDA/HIP device mode.

-fheinous-gnu-extensions

-flat_namespace

-fopenmp-targets=<arg1>,<arg2>...

Specify comma-separated list of triples OpenMP offloading targets to be supported

-force_cpusubtype_ALL

-force_flat_namespace

-force_load <arg>

-fplugin-arg-<name>-<arg>

Pass <arg> to plugin <name>

-framework <arg>

-frtlib-add-rpath, **-fno-rtlib-add-rpath**

Add -rpath with architecture-specific resource directory to the linker flags

-fsanitize-system-ignorelist=<arg>, **-fsanitize-system-blacklist**=<arg>

Path to system ignorelist file for sanitizers

-fshow-skipped-includes

#include files may be "skipped" due to include guard optimization

or #pragma once. This flag makes -H show also such includes.

-fsystem-module

Build this module as a system module. Only used with -emit-module

-fuse-cuid=<arg>

Method to generate ID’s for compilation units for single source offloading languages CUDA and HIP: ‘hash’ (ID’s generated by hashing file path and command line options) | ‘random’ (ID’s generated as random numbers) | ‘none’ (disabled). Default is ‘hash’. This option will be overridden by option ‘-cuid=[ID]’ if it is specified.

--gcc-toolchain=<arg>

Search for GCC installation in the specified directory on targets which commonly use GCC. The directory usually contains ‘lib{,32,64}/gcc{,-cross}/$triple’ and ‘include’. If specified, sysroot is skipped for GCC detection. Note: executables (e.g. ld) used by the compiler are not overridden by the selected GCC installation

-gcodeview

Generate CodeView debug information

-gcodeview-ghash, **-gno-codeview-ghash**

Emit type record hashes in a .debug$H section

-gen-reproducer=<arg>, **-fno-crash-diagnostics** (equivalent to -gen-reproducer=off)

Emit reproducer on (option: off, crash (default), error, always)

--gpu-instrument-lib=<arg>

Instrument device library for HIP, which is a LLVM bitcode containing __cyg_profile_func_enter and __cyg_profile_func_exit

--gpu-max-threads-per-block=<arg>

Default max threads per block for kernel launch bounds for HIP

-headerpad_max_install_names<arg>

-help, **--help**, **/help<arg>**, **-help<arg>**, **--help<arg>**

Display available options

--help-hidden

Display help for hidden options

--hip-link

Link clang-offload-bundler bundles for HIP

--hip-version=<arg>

HIP version in the format of major.minor.patch

-ibuiltininc

Enable builtin #include directories even when -nostdinc is used before or after -ibuiltininc. Using -nobuiltininc after the option disables it

-image_base <arg>

-index-header-map

Make the next included directory (-I or -F) an indexer header map

-init <arg>

-install_name <arg>

-interface-stub-version=<arg>

-keep_private_externs

-lazy_framework <arg>

-lazy_library <arg>

-mbig-endian, **-EB**

-mdataimported

All variables can be treated as imported

-mdataimported=<arg1>,<arg2>...

Specifies which variables can be treated as imported

-mdatalocal

All variables can be treated as local

-mdatalocal=<arg1>,<arg2>...

Specifies which variables can be treated as local

-menable-unsafe-fp-math

Allow unsafe floating-point math optimizations which may decrease precision

-mharden-sls=<arg>

Select straight-line speculation hardening scope (ARM/AArch64/X86 only). <arg> must be: all, none, retbr(ARM/AArch64), blr(ARM/AArch64), comdat(ARM/AArch64), nocomdat(ARM/AArch64), return(X86), indirect-jmp(X86)

--migrate

Run the migrator

-mios-simulator-version-min=<arg>, **-miphonesimulator-version-min**=<arg>

-mlinker-version=<arg>

-mlittle-endian, **-EL**

-mllvm <arg>

Additional arguments to forward to LLVM's option processing

-mmlir <arg>

Additional arguments to forward to MLIR's option processing

-module-dependency-dir <arg>

Directory to dump module dependencies to

-mtvos-simulator-version-min=<arg>, **-mappletvsimulator-version-min**=<arg>

-multi_module

-multiply_defined <arg>

-multiply_defined_unused <arg>

-mwatchos-simulator-version-min=<arg>, **-mwatchsimulator-version-min**=<arg>

--no-cuda-version-check

Don't error out if the detected version of the CUDA install is too low for the requested CUDA gpu architecture.

-no-hip-rt

Do not link against HIP runtime libraries

-no-integrated-cpp, **--no-integrated-cpp**

-no_dead_strip_inits_and_terms

-nobuiltininc

Disable builtin #include directories

-nodefaultlibs

-nodriverkitlib

-nofixprebinding

-nogpuinc, **-nocudainc**

Do not add include paths for CUDA/HIP and do not include the default CUDA/HIP wrapper headers

-nogpulib, **-nocudalib**

Do not link device library for CUDA/HIP device compilation

-nohipwrapperinc

Do not include the default HIP wrapper headers and include paths

-nolibc

-nomultidefs

-nopie, **-no-pie**

-noprebind

-noprofilelib

-noseglinkedit

-nostdinc, **--no-standard-includes**

-nostdinc++

Disable standard #include directories for the C++ standard library

-nostdlib++

-nostdlibinc

-o<file>, **/Fo<arg>**, **-Fo<arg>**, **--output** <arg>, **--output**=<arg>

Write output to <file>

-objcmt-allowlist-dir-path=<arg>, **-objcmt-white-list-dir-path**=<arg>, **-objcmt-whitelist-dir-path**=<arg>

Only modify files with a filename contained in the provided directory path

-objcmt-atomic-property

Make migration to ‘atomic’ properties

-objcmt-migrate-all

Enable migration to modern ObjC

-objcmt-migrate-annotation

Enable migration to property and method annotations

-objcmt-migrate-designated-init

Enable migration to infer NS_DESIGNATED_INITIALIZER for initializer methods

-objcmt-migrate-instancetype

Enable migration to infer instancetype for method result type

-objcmt-migrate-literals

Enable migration to modern ObjC literals

-objcmt-migrate-ns-macros

Enable migration to NS_ENUM/NS_OPTIONS macros

-objcmt-migrate-property

Enable migration to modern ObjC property

-objcmt-migrate-property-dot-syntax

Enable migration of setter/getter messages to property-dot syntax

-objcmt-migrate-protocol-conformance

Enable migration to add protocol conformance on classes

-objcmt-migrate-readonly-property

Enable migration to modern ObjC readonly property

-objcmt-migrate-readwrite-property

Enable migration to modern ObjC readwrite property

-objcmt-migrate-subscripting

Enable migration to modern ObjC subscripting

-objcmt-ns-nonatomic-iosonly

Enable migration to use NS_NONATOMIC_IOSONLY macro for setting property's 'atomic' attribute

-objcmt-returns-innerpointer-property

Enable migration to annotate property with NS_RETURNS_INNER_POINTER

-object

-object-file-name=<file>, **-object-file-name** <arg>

Set the output <file> for debug infos

--offload-arch=<arg>, **--cuda-gpu-arch**=<arg>, **--no-offload-arch**=<arg>

CUDA offloading device architecture (e.g. sm_35), or HIP offloading target ID in the form of a device architecture followed by target ID features delimited by a colon. Each target ID feature is a pre-defined string followed by a plus or minus sign (e.g. gfx908:xnack+:sramecc-). May be specified more than once.

--offload-device-only, **--cuda-device-only**

Only compile for the offloading device.

--offload-host-device, **--cuda-compile-host-device**

Only compile for the offloading host.

--offload-host-only, **--cuda-host-only**

Only compile for the offloading host.

--offload=<arg1>,<arg2>...

Specify comma-separated list of offloading target triples (CUDA and HIP only)

-p, **--profile**

-pagezero_size<arg>

-pg

Enable mcount instrumentation

-pipe, **--pipe**

Use pipes between commands, when possible

-prebind

-prebind_all_twolevel_modules

-preload

--print-diagnostic-categories

-print-effective-triple, **--print-effective-triple**

Print the effective target triple

-print-file-name=<file>, **--print-file-name**=<file>, **--print-file-name** <arg>

Print the full library path of <file>

-print-ivar-layout

Enable Objective-C Ivar layout bitmap print trace

-print-libgcc-file-name, **--print-libgcc-file-name**

Print the library path for the currently used compiler runtime library ("libgcc.a" or "libclang_rt.builtins.*.a")

-print-multi-directory, **--print-multi-directory**

-print-multi-lib, **--print-multi-lib**

-print-multiarch, **--print-multiarch**

Print the multiarch target triple

-print-prog-name=<name>, **--print-prog-name**=<name>, **--print-prog-name** <arg>

Print the full program path of <name>

-print-resource-dir, **--print-resource-dir**

Print the resource directory pathname

-print-rocm-search-dirs, **--print-rocm-search-dirs**

Print the paths used for finding ROCm installation

-print-runtime-dir, **--print-runtime-dir**

Print the directory pathname containing clangs runtime libraries

-print-search-dirs, **--print-search-dirs**

Print the paths used for finding libraries and programs

-print-target-triple, **--print-target-triple**

Print the normalized target triple

-print-targets, **--print-targets**

Print the registered targets

-private_bundle

--product-name=<arg>

-pthread, **-no-pthread**

Support POSIX threads in generated code

-pthreads

-read_only_relocs <arg>

-relocatable-pch, **--relocatable-pch**

Whether to build a relocatable precompiled header

-remap

-rewrite-legacy-objc

Rewrite Legacy Objective-C source to C++

-rtlib=<arg>, **--rtlib**=<arg>, **--rtlib** <arg>

Compiler runtime library to use

-save-stats=<arg>, **--save-stats**=<arg>, **-save-stats** (equivalent to -save-stats=cwd), **--save-stats** (equivalent to -save-stats=cwd)

Save llvm statistics.

-save-temps=<arg>, **--save-temps**=<arg>, **-save-temps** (equivalent to -save-temps=cwd), **--save-temps** (equivalent to -save-temps=cwd)

Save intermediate compilation results.

-sectalign <arg1> <arg2> <arg3>

-sectcreate <arg1> <arg2> <arg3>

-sectobjectsymbols <arg1> <arg2>

-sectorder <arg1> <arg2> <arg3>

-seg1addr<arg>

-seg_addr_table <arg>

-seg_addr_table_filename <arg>

-segaddr <arg1> <arg2>

-segcreate <arg1> <arg2> <arg3>

-seglinkedit

-segprot <arg1> <arg2> <arg3>

-segs_read_<arg>

-segs_read_only_addr <arg>

-segs_read_write_addr <arg>

-serialize-diagnostics <arg>, **--serialize-diagnostics** <arg>

Serialize compiler diagnostics to a file

-shared-libgcc

-shared-libsan, **-shared-libasan**

Dynamically link the sanitizer runtime

-single_module

-slm-auth=<file>, **--slm-auth**=<file>

The path of the authorization file

-slm-dir=<dir>, **--slm-dir**=<dir>

The directory of the SLM tag file

-slm-limit=<limit>, **--slm-limit**=<limit>

The maximum number of bytes that each tag file is allowed to occupy

-slm-period=<period>, **--slm-period**=<period>

The number of seconds that each metric covers

-slm-timeout=<timeout>, **--slm-timeout**=<timeout>

The minimum number of seconds that the daemon must wait before terminating

--start-no-unused-arguments

Don’t emit warnings about unused arguments for the following arguments

-static-libgcc

-static-libsan

Statically link the sanitizer runtime

-static-libstdc++

-static-openmp

Use the static host OpenMP runtime while linking.

-std-default=<arg>

-stdlib=<arg>, **--stdlib**=<arg>, **--stdlib** <arg>

C++ standard library to use. <arg> must be ‘libc++’, ‘libstdc++’ or ‘platform’.

-sub_library<arg>

-sub_umbrella<arg>

--sysroot=<arg>, **--sysroot** <arg>

--target-help

--target=<arg>, **-target** <arg>

Generate code for the given target

-time

Time individual commands

-traditional, **--traditional**

-traditional-cpp, **--traditional-cpp**

Enable some traditional CPP emulation

-twolevel_namespace

-twolevel_namespace_hints

-umbrella <arg>

-unexported_symbols_list <arg>

-unwindlib=<arg>, **--unwindlib**=<arg>

Unwind library to use. <arg> must be ‘libgcc’, ‘unwindlib’ or ‘platform’.

-v, **--verbose**

Show commands to run and use verbose output

--verify-debug-info

Verify the binary representation of debug output

--version

Print version information

-w, **--no-warnings**

Suppress all warnings

-weak-l<arg>

-weak_framework <arg>

-weak_library <arg>

-weak_reference_mismatches <arg>

-whatsloaded

-why_load, **-whyload**

-working-directory<arg>, **-working-directory**=<arg>

Resolve file paths relative to the specified directory

-x<language>, **--language** <arg>, **--language**=<arg>

Treat subsequent input files as having type <language>

-y<arg>

Actions

The action to perform on the input.

-E, **--preprocess**

Only run the preprocessor

-S, **--assemble**

Only run preprocess and compilation steps

-c, **--compile**

Only run preprocess, compile, and assemble steps

-emit-interface-stubs

Generate Interface Stub Files.

-emit-llvm

Use the LLVM representation for assembler and object files

-emit-merged-ifs

Generate Interface Stub Files, emit merged text not binary.

-extract-api

Extract API information

-fsyntax-only

-module-file-info

Provide information about a particular module file

--precompile

Only precompile the input

-rewrite-objc

Rewrite Objective-C source to C++

-verify-pch

Load and verify that a pre-compiled header file is not stale

Compilation flags

Flags controlling the behavior of Clang during compilation. These flags have no effect during actions that do not perform compilation.

-Xassembler <arg>

Pass <arg> to the assembler

-Xclang <arg>

Pass <arg> to the clang compiler

-Xopenmp-target <arg>

Pass <arg> to the target offloading toolchain.

-Xopenmp-target=<triple> <arg>

Pass <arg> to the target offloading toolchain identified by <triple>.

-ansi, **--ansi**

-fc++-abi=<arg>

C++ ABI to use. This will override the target C++ ABI.

-fclang-abi-compat=<version>

Attempt to match the ABI of Clang <version>. <version> must be '<major>.<minor>' or 'latest'.

-fcomment-block-commands=<arg>,<arg2>...

Treat each comma separated argument in <arg> as a documentation comment block command

-fcomplete-member-pointers, **-fno-complete-member-pointers**

Require member pointer base types to be complete if they would be significant under the Microsoft ABI

-fcrash-diagnostics-dir=<dir>

Put crash-report files in <dir>

-fdeclspec, **-fno-declspec**

Allow __declspec as a keyword

-fdepfile-entry=<arg>

-fdiagnostics-fixit-info, **-fno-diagnostics-fixit-info**

-fdiagnostics-format=<arg>

-fdiagnostics-parseable-fixits

Print fix-its in machine parseable form

-fdiagnostics-print-source-range-info

Print source range spans in numeric form

-fdiagnostics-show-category=<arg>

-fdiscard-value-names, **-fno-discard-value-names**

Discard value names in LLVM IR

-fexperimental-relative-c++-abi-vtables, **-fno-experimental-relative-c++-abi-vtables**

Use the experimental C++ class ABI for classes with virtual tables

-fexperimental-strict-floating-point

Enables experimental strict floating point in LLVM.

-ffine-grained-bitfield-accesses, **-fno-fine-grained-bitfield-accesses**

Use separate accesses for consecutive bitfield runs with legal widths and alignments.

-fglobal-isel, **-fexperimental-isel**, **-fno-global-isel**

Enables the global instruction selector

-finline-functions, **-fno-inline-functions**

Inline suitable functions

-finline-hint-functions

Inline functions which are (explicitly or implicitly) marked inline

-fno-legacy-pass-manager, **-fexperimental-new-pass-manager**

-fno-sanitize-ignorelist, **-fno-sanitize-blacklist**

Don’t use ignorelist file for sanitizers

-fparse-all-comments

-frandomize-layout-seed-file=<file>

File holding the seed used by the randomize structure layout feature

-frandomize-layout-seed=<seed>

The seed used by the randomize structure layout feature

-frecord-command-line, **-fno-record-command-line**, **-frecord-gcc-switches**

-fsanitize-address-destructor=<arg>

Set destructor type used in ASan instrumentation. <arg> must be ‘none’ or ‘global’.

-fsanitize-address-field-padding=<arg>

Level of field padding for AddressSanitizer

-fsanitize-address-globals-dead-stripping,
-fno-sanitize-address-globals-dead-stripping

Enable linker dead stripping of globals in AddressSanitizer

-fsanitize-address-outline-instrumentation,
-fno-sanitize-address-outline-instrumentation

Always generate function calls for address sanitizer instrumentation

-fsanitize-address-poison-custom-array-cookie, **-fno-sanitize-address-poison-custom-array-cookie**

Enable poisoning array cookies when using custom operator new[] in AddressSanitizer

-fsanitize-address-use-after-return=<mode>

Select the mode of detecting stack use-after-return in AddressSanitizer. <mode> must be 'never', 'runtime' or 'always'.

-fsanitize-address-use-after-scope, **-fno-sanitize-address-use-after-scope**

Enable use-after-scope detection in AddressSanitizer

-fsanitize-address-use-odr-indicator, **-fno-sanitize-address-use-odr-indicator**

Enable ODR indicator globals to avoid false ODR violation reports in partially sanitized programs at the cost of an increase in binary size

-fsanitize-cfi-canonical-jump-tables, **-fno-sanitize-cfi-canonical-jump-tables**

Make the jump table addresses canonical in the symbol table

-fsanitize-cfi-cross-dso, **-fno-sanitize-cfi-cross-dso**

Enable control flow integrity (CFI) checks for cross-DSO calls.

-fsanitize-cfi-icall-generalize-pointers

Generalize pointers in CFI indirect call type signature checks

-fsanitize-coverage-allowlist=<arg>, **-fsanitize-coverage-whitelist**=<arg>

Restrict sanitizer coverage instrumentation exclusively to modules and functions that match the provided special case list, except the blocked ones

-fsanitize-coverage-ignorelist=<arg>, **-fsanitize-coverage-blacklist**=<arg>

Disable sanitizer coverage instrumentation for modules and functions that match the provided special case list, even the allowed ones

-fsanitize-coverage=<arg1>,<arg2>..., **-fno-sanitize-coverage**=<arg1>,<arg2>...

Specify the type of coverage instrumentation for Sanitizers

-fsanitize-hwaddress-abi=<arg>

Select the HWAddressSanitizer ABI to target (interceptor or platform, default interceptor). This option is currently unused.

-fsanitize-hwaddress-experimental-aliasing, **-fno-sanitize-hwaddress-experimental-aliasing**

Enable aliasing mode in HWAddressSanitizer

-fsanitize-ignorelist=<arg>, **-fsanitize-blacklist**=<arg>

Path to ignorelist file for sanitizers

-fsanitize-link-c++-runtime, **-fno-sanitize-link-c++-runtime**

-fsanitize-link-runtime, **-fno-sanitize-link-runtime**

-fsanitize-memory-track-origins, **-fno-sanitize-memory-track-origins**

Enable origins tracking in MemorySanitizer

-fsanitize-memory-track-origins=<arg>

Enable origins tracking in MemorySanitizer

-fsanitize-memory-use-after-dtor, **-fno-sanitize-memory-use-after-dtor**

Enable use-after-destroy detection in MemorySanitizer

-fsanitize-memtag-mode=<arg>

Set default MTE mode to ‘sync’ (default) or ‘async’

-fsanitize-minimal-runtime, **-fno-sanitize-minimal-runtime**

-fsanitize-recover=<arg1>,<arg2>..., **-fno-sanitize-recover**=<arg1>,<arg2>...,
-fsanitize-recover (equivalent to -fsanitize-recover=all)

Enable recovery for specified sanitizers

-fsanitize-stats, **-fno-sanitize-stats**

Enable sanitizer statistics gathering.

-fsanitize-thread-atomics, **-fno-sanitize-thread-atomics**

Enable atomic operations instrumentation in ThreadSanitizer (default)

-fsanitize-thread-func-entry-exit, **-fno-sanitize-thread-func-entry-exit**

Enable function entry/exit instrumentation in ThreadSanitizer (default)

-fsanitize-thread-memory-access, **-fno-sanitize-thread-memory-access**

Enable memory access instrumentation in ThreadSanitizer (default)

-fsanitize-trap=<arg1>,<arg2>..., **-fno-sanitize-trap**=<arg1>,<arg2>...,
-fsanitize-trap (equivalent to -fsanitize-trap=all),
-fsanitize-undefined-trap-on-error (equivalent to -fsanitize-trap=undefined)

Enable trapping for specified sanitizers

-fsanitize-undefined-strip-path-components=<number>

Strip (or keep only, if negative) a given number of path components when emitting check metadata.

-fsanitize=<check>,<arg2>..., **-fno-sanitize**=<arg1>,<arg2>...

Turn on runtime checks for various forms of undefined or suspicious behavior. See user manual for available checks

-moutline, **-mno-outline**

Enable function outlining (AArch64 only)

-moutline-atomics, **-mno-outline-atomics**

Generate local calls to out-of-line atomic operations

--param <arg>, **--param**=<arg>

-print-supported-cpus, **--print-supported-cpus**, **-mcpu**=?, **-mtune**=?

Print supported cpu models for the given target (if target is not specified, it will print the supported cpus for the default target)

-std=<arg>, **--std**=<arg>, **--std** <arg>

Language standard to compile for

Preprocessor flags

Flags controlling the behavior of the Clang preprocessor.

-C, **--comments**

Include comments in preprocessed output

-CC, **--comments-in-macros**

Include comments from within macros in preprocessed output

-D<macro>=<value>, **--D<arg>**, **/D<arg>**, **-D<arg>**, **--define-macro** <arg>,
--define-macro=<arg>

Define <macro> to <value> (or 1 if <value> omitted)

-H, **--trace-includes**

Show header includes and nesting depth

-P, --no-line-commands

Disable linemarker output in -E mode

-U<macro>, --undefine-macro <arg>, **--undefine-macro**=<arg>

Undefine macro <macro>

-Wp,<arg>,<arg2>...

Pass the comma separated arguments in <arg> to the preprocessor

-Xpreprocessor <arg>

Pass <arg> to the preprocessor

Include path management

Flags controlling how #includes are resolved to files.

-I<dir>, --include-directory <arg>, **--include-directory**=<arg>

Add directory to include search path. For C++ inputs, if there are multiple -I options, these directories are searched in the order they are given before the standard system directories are searched. If the same directory is in the SYSTEM include search paths, for example if also specified with -isystem, the -I option will be ignored

-I-, --include-barrier

Restrict all prior -I flags to double-quoted inclusion and remove current directory from include path

--amdgpu-arch-tool=<arg>

Tool used for detecting AMD GPU arch in the system.

--cuda-path-ignore-env

Ignore environment variables to detect CUDA installation

--cuda-path=<arg>

CUDA installation path

-cxx-isystem<directory>

Add directory to the C++ SYSTEM include search path

-fbuild-session-file=<file>

Use the last modification time of <file> as the build session timestamp

-fbuild-session-timestamp=<time since Epoch in seconds>

Time when the current build session started

-fmodule-file=\[<name>=\]<file>

Specify the mapping of module name to precompiled module file, or load a module file if name is omitted.

-fmodules-cache-path=<directory>

Specify the module cache path

-fmodules-disable-diagnostic-validation

Disable validation of the diagnostic options when loading the module

-fmodules-prune-after=<seconds>

Specify the interval (in seconds) after which a module file will be considered unused

-fmodules-prune-interval=<seconds>

Specify the interval (in seconds) between attempts to prune the module cache

-fmodules-user-build-path <directory>

Specify the module user build path

-fmodules-validate-once-per-build-session

Don’t verify input files for the modules if the module has been successfully validated or loaded during this build session

-fmodules-validate-system-headers, **-fno-modules-validate-system-headers**

Validate the system headers that a module depends on when loading the module

-fprebuilt-module-path=<directory>

Specify the prebuilt module path

--hip-path=<arg>

HIP runtime installation path, used for finding HIP version and adding HIP include path.

-idirafter<arg>, **--include-directory-after** <arg>, **--include-directory-after**=<arg>

Add directory to AFTER include search path

-iframework<arg>

Add directory to SYSTEM framework search path

-iframeworkwithsysroot<directory>

Add directory to SYSTEM framework search path, absolute paths are relative to -isysroot

-imacros<file>, **--imacros<file>**, **--imacros**=<arg>

Include macros from file before parsing

-include<file>, **--include<file>**, **--include**=<arg>

Include file before parsing

-include-pch <file>

Include precompiled header file

-iprefix<dir>, **--include-prefix** <arg>, **--include-prefix**=<arg>

Set the -iwithprefix/-iwithprefixbefore prefix

-iquote<directory>

Add directory to QUOTE include search path

-isysroot<dir>

Set the system root directory (usually /)

-isystem<directory>

Add directory to SYSTEM include search path

-isystem-after<directory>

Add directory to end of the SYSTEM include search path

-ivfsoverlay<arg>

Overlay the virtual filesystem described by file over the real file system

-iwithprefix<dir>, **--include-with-prefix** <arg>, **--include-with-prefix-after** <arg>, **--include-with-prefix-after**=<arg>, **--include-with-prefix**=<arg>

Set directory to SYSTEM include search path with prefix

-iwithprefixbefore<dir>, **--include-with-prefix-before** <arg>, **--include-with-prefix-before**=<arg>

Set directory to include search path with prefix

-iwithsysroot<directory>

Add directory to SYSTEM include search path, absolute paths are relative to -isysroot

--libomptarget-amdgpu-bc-path=<arg>, **--libomptarget-amdgcn-bc-path**=<arg>

Path to libomptarget-amdgcn bitcode library

--libomptarget-nvptx-bc-path=<arg>

Path to libomptarget-nvptx bitcode library

--ptxas-path=<arg>

Path to ptxas (used for compiling CUDA code)

--rocm-path=<arg>

ROCm installation path, used for finding and automatically linking required bitcode libraries.

-stdlib++-isystem<directory>

Use directory as the C++ standard library include path

--system-header-prefix=<prefix>, **--no-system-header-prefix**=<prefix>, **--system-header-prefix** <arg>

Treat all #include paths starting with <prefix> as including a system header.

Dependency file generation

Flags controlling generation of a dependency file for `make`-like build systems.

-M, --dependencies

Like -MD, but also implies -E and writes to stdout by default

-MD, --write-dependencies

Write a depfile containing user and system headers

-MF<file>

Write depfile output from -MMD, -MD, -MM, or -M to <file>

-MG, --print-missing-file-dependencies

Add missing headers to depfile

-MJ<arg>

Write a compilation database entry per input

-MM, --user-dependencies

Like -MMD, but also implies -E and writes to stdout by default

-MMD, --write-user-dependencies

Write a depfile containing user headers

-MP

Create phony target for each dependency (other than main file)

-MQ<arg>

Specify name of main file output to quote in depfile

-MT<arg>

Specify name of main file output in depfile

-MV

Use NMake/Jom format for the depfile

Dumping preprocessor state

Flags allowing the state of the preprocessor to be dumped in various ways.

-d

-d<arg>

-dD

Print macro definitions in -E mode in addition to normal output

-dI

Print include directives in -E mode in addition to normal output

-dM

Print macro definitions in -E mode instead of normal output

Diagnostic flags

Flags controlling which warnings, errors, and remarks Clang will generate. See the full list of warning and remark flags.

-R<remark>

Enable the specified remark

-Rpass-analysis=<arg>

Report transformation analysis from optimization passes whose name matches the given POSIX regular expression

-Rpass-missed=<arg>

Report missed transformations by optimization passes whose name matches the given POSIX regular expression

-Rpass=<arg>

Report transformations performed by optimization passes whose name matches the given POSIX regular expression

-W<warning>, **--extra-warnings**, **--warn-<arg>**, **--warn-**=<arg>

Enable the specified warning

-Wdeprecated, **-Wno-deprecated**

Enable warnings for deprecated constructs and define __DEPRECATED

-Wframe-larger-than=<arg>, **-Wframe-larger-than**

-Wnonportable-cfstrings<arg>, **-Wno-nonportable-cfstrings<arg>**

Target-independent compilation options

-fPIC, **-fno-PIC**

-fPIE, **-fno-PIE**

-faccess-control, **-fno-access-control**

-faddrsig, **-fno-addrsig**

Emit an address-significance table

-falign-functions, **-fno-align-functions**

-falign-functions=<arg>

-falign-loops=<N>

N must be a power of two. Align loops to the boundary

-faligned-allocation, **-faligned-new**, **-fno-aligned-allocation**

Enable C++17 aligned allocation functions

-fallow-editor-placeholders, **-fno-allow-editor-placeholders**

Treat editor placeholders as valid source code

-fallow-unsupported

-faltivec, **-fno-altivec**

-faltivec-src-compat=<arg>

Source-level compatibility for Altivec vectors (for PowerPC targets). This includes results of vector comparison (scalar for ‘xl’, vector for ‘gcc’) as well as behavior when initializing with a scalar (splatting for ‘xl’, element zero only for ‘gcc’). For ‘mixed’, the compatibility is as ‘gcc’ for ‘vector bool/vector pixel’ and as ‘xl’ for other types. Current default is ‘mixed’. <arg> must be ‘mixed’, ‘gcc’ or ‘xl’.

-fansi-escape-codes

Use ANSI escape codes for diagnostics

-fapple-kext, **-findirect-virtual-calls**, **-fterminated-vtables**

Use Apple’s kernel extensions ABI

-fapple-link-rtlib

Force linking the clang builtins runtime library

-fapple-pragma-pack, **-fno-apple-pragma-pack**

Enable Apple gcc-compatible #pragma pack handling

-fapplication-extension, **-fno-application-extension**

Restrict code to those available for App Extensions

-fapprox-func, **-fno-approx-func**

Allow certain math function calls to be replaced with an approximately equivalent calculation

-fasm, **-fno-asm**

-fasm-blocks, **-fno-asm-blocks**

-fassociative-math, **-fno-associative-math**

-fassume-sane-operator-new, **-fno-assume-sane-operator-new**

-fast

-fastcp

-fastf

-fasync-exceptions, **-fno-async-exceptions**

Enable EH Asynchronous exceptions

-fasynchronous-unwind-tables, **-fno-asynchronous-unwind-tables**

-fautolink, **-fno-autolink**

-fbasic-block-sections=<arg>

Generate labels for each basic block or place each basic block or a subset of basic blocks in its own section. <arg> must be ‘all’, ‘labels’, ‘none’ or ‘list=’.

-fbinutils-version=<major.minor>

Produced object files can use all ELF features supported by this binutils version and newer. If -fno-integrated-as is specified, the generated assembly will consider GNU as support. ‘none’ means that all ELF features can be used, regardless of binutils support. Defaults to 2.26.

-fblocks, **-fno-blocks**

Enable the ‘blocks’ language feature

-fbootclasspath=<arg>, **--bootclasspath** <arg>, **--bootclasspath**=<arg>

-fborland-extensions, **-fno-borland-extensions**

Accept non-standard constructs supported by the Borland compiler

-fbracket-depth=<arg>

-fbuiltin, **-fno-builtin**

-fbuiltin-module-map

Load the clang builtins module map file.

-fc++-static-destructors, **-fno-c++-static-destructors**

-fcaret-diagnostics, **-fno-caret-diagnostics**

-fcf-protection=<arg>, **-fcf-protection** (equivalent to -fcf-protection=full)

Instrument control-flow architecture protection. <arg> must be ‘return’, ‘branch’, ‘full’ or ‘none’.

-fcf-runtime-abi=<arg>
<arg> must be ‘unspecified’, ‘standalone’, ‘objc’, ‘swift’, ‘swift-5.0’, ‘swift-4.2’ or ‘swift-4.1’.

-fchar8_t, **-fno-char8_t**

Enable C++ builtin type char8_t

-fclasspath=<arg>, **--CLASSPATH** <arg>, **--CLASSPATH**=<arg>, **--classpath** <arg>, **--classpath**=<arg>

-fcolor-diagnostics, **-fdiagnostics-color**, **-fno-color-diagnostics**

Enable colors in diagnostics

-fcommon, **-fno-common**

Place uninitialized global variables in a common block

-fcompile-resource=<arg>, **--resource** <arg>, **--resource**=<arg>

-fconstant-cfstrings, **-fno-constant-cfstrings**

-fconstant-string-class=<arg>

-fconstexpr-backtrace-limit=<arg>

-fconstexpr-depth=<arg>

-fconstexpr-steps=<arg>

-fconvergent-functions

Assume functions may be convergent

-fcoroutines-ts, **-fno-coroutines-ts**

Enable support for the C++ Coroutines TS

-fcoverage-compilation-dir=<arg>

The compilation directory to embed in the coverage mapping.

-fcoverage-mapping, **-fno-coverage-mapping**

Generate coverage mapping to enable code coverage analysis

-fcoverage-prefix-map=<arg>

remap file source paths in coverage mapping

-fcreate-profile

-fcs-profile-generate

Generate instrumented code to collect context sensitive execution counts into default.profraw (overridden by LLVM_PROFILE_FILE env var)

-fcs-profile-generate=<directory>

Generate instrumented code to collect context sensitive execution counts into <directory>/default.profraw (overridden by LLVM_PROFILE_FILE env var)

-fcuda-approx-transcendentals, **-fno-cuda-approx-transcendentals**

Use approximate transcendental functions

-fcuda-short-ptr, **-fno-cuda-short-ptr**

Use 32-bit pointers for accessing const/local/shared address spaces

-fcxx-exceptions, **-fno-cxx-exceptions**

Enable C++ exceptions

-fcxx-modules, **-fno-cxx-modules**

Enable modules for C++

-fdata-sections, **-fno-data-sections**

Place each data in its own section

-fdebug-compilation-dir=<arg>, **-fdebug-compilation-dir** <arg>

The compilation directory to embed in the debug info

-fdebug-default-version=<arg>

Default DWARF version to use, if a -g option caused DWARF debug info to be produced

-fdebug-info-for-profiling, **-fno-debug-info-for-profiling**

Emit extra debug info to make sample profile more accurate

-fdebug-macro, **-fno-debug-macro**

Emit macro debug information

-fdebug-pass-arguments

-fdebug-pass-structure

-fdebug-prefix-map=<arg>

remap file source paths in debug info

-fdebug-ranges-base-address, **-fno-debug-ranges-base-address**

Use DWARF base address selection entries in .debug_ranges

-fdebug-types-section, **-fno-debug-types-section**

Place debug types in their own section (ELF Only)

-fdelayed-template-parsing, **-fno-delayed-template-parsing**

Parse templated function definitions at the end of the translation unit

-fdelete-null-pointer-checks, **-fno-delete-null-pointer-checks**

Treat usage of null pointers as undefined behavior (default)

-fdenormal-fp-math=<arg>

-fdiagnostics-absolute-paths

Print absolute paths in diagnostics

-fdiagnostics-color=<arg>

-fdiagnostics-hotness-threshold=<value>

Prevent optimization remarks from being output if they do not have at least this profile count. Use ‘auto’ to apply the threshold from profile summary

-fdiagnostics-misexpect-tolerance=<value>

Prevent misexpect diagnostics from being output if the profile counts are within N% of the expected.

-fdiagnostics-show-hotness, **-fno-diagnostics-show-hotness**

Enable profile hotness information in diagnostic line

-fdiagnostics-show-note-include-stack, **-fno-diagnostics-show-note-include-stack**

Display include stacks for diagnostic notes

-fdiagnostics-show-option, **-fno-diagnostics-show-option**

Print option name with mappable diagnostics

-fdiagnostics-show-template-tree

Print a template comparison tree for differing templates

-fdigraphs, **-fno-digraphs**

Enable alternative token representations ‘<:’, ‘:>’, ‘<%’, ‘%>’, ‘%:’, ‘%:%:’ (default)

-fdirect-access-external-data, **-fno-direct-access-external-data**

Don’t use GOT indirection to reference external data symbols

-fdirectives-only, **-fno-directives-only**

-fdollars-in-identifiers, **-fno-dollars-in-identifiers**

Allow ‘$’ in identifiers

-fdouble-square-bracket-attributes, **-fno-double-square-bracket-attributes**

Enable ‘[[]]’ attributes in all C and C++ language modes

-fdump-margin-seq-filetag

Print out margin sequence and file tag information

-fdwarf-directory-asm, **-fno-dwarf-directory-asm**

-fdwarf-exceptions

Use DWARF style exceptions

-felide-constructors, **-fno-elide-constructors**

-feliminate-unused-debug-symbols, **-fno-eliminate-unused-debug-symbols**

-feliminate-unused-debug-types, **-fno-eliminate-unused-debug-types**

Do not emit debug info for defined but unused types

-fembed-bitcode=<option>, **-fembed-bitcode** (equivalent to -fembed-bitcode=all), **-fembed-bitcode-marker** (equivalent to -fembed-bitcode=marker)

Embed LLVM bitcode. <option> must be ‘off’, ‘all’, ‘bitcode’ or ‘marker’.

-fembed-offload-object=<arg>

Embed Offloading device-side binary into host object file as a section.

-femit-all-decls

Emit all declarations, even if unused

-femulated-tls, **-fno-emulated-tls**

Use emutls functions to access thread_local variables

-fenable-matrix

Enable matrix data type and related builtin functions

-fencoding=<arg>, **--encoding** <arg>, **--encoding**=<arg>

-ferror-limit=<arg>

-fescaping-block-tail-calls, **-fno-escaping-block-tail-calls**

-fexceptions, **-fno-exceptions**

Enable support for exception handling

-fexec-charset=<arg>

-fexperimental-new-constant-interpreter

Enable the experimental new constant interpreter

-fextdirs=<arg>, **--extdirs** <arg>, **--extdirs**=<arg>

-fextend-arguments=<arg>

Controls how scalar integer arguments are extended in calls to unprototyped and varargs functions. <arg> must be ‘32’ or ‘64’.

-ffast-math, -fno-fast-math

Allow aggressive, lossy floating-point optimizations

-ffile-compilation-dir=<arg>

The compilation directory to embed in the debug info and coverage mapping.

-ffile-prefix-map=<arg>

remap file source paths in debug info, predefined preprocessor macros and __builtin_FILE(). Implies -ffile-reproducible.

-ffile-reproducible, -fno-file-reproducible

Use the target’s platform-specific path separator character when expanding the __FILE__ macro

-ffinite-loops, -fno-finite-loops

Assume all loops are finite.

-ffinite-math-only, -fno-finite-math-only

-ffixed-point, -fno-fixed-point

Enable fixed point types

-ffor-scope, -fno-for-scope

-fforce-dwarf-frame, -fno-force-dwarf-frame

Always emit a debug frame section

-fforce-emit-vtables, -fno-force-emit-vtables

Emits more virtual tables to improve devirtualization

-fforce-enable-int128, -fno-force-enable-int128

Enable support for int128_t type

-ffp-contract=<arg>

Form fused FP ops (e.g. FMAs): fast (fuses across statements disregarding pragmas) | on (only fuses in the same statement unless dictated by pragmas) | off (never fuses) | fast-honor-pragmas (fuses across statements unless diectated by pragmas). Default is ‘fast’ for CUDA, ‘fast-honor-pragmas’ for HIP, and ‘on’ otherwise. <arg> must be ‘fast’, ‘on’, ‘off’ or ‘fast-honor-pragmas’.

-ffp-eval-method=<arg>

Specifies the evaluation method to use for floating-point arithmetic. <arg> must be ‘source’, ‘double’ or ‘extended’.

-ffp-exception-behavior=<arg>

Specifies the exception behavior of floating-point operations. <arg> must be ‘ignore’, ‘maytrap’ or ‘strict’.

-ffp-model=<arg>

Controls the semantics of floating-point calculations.

-ffreestanding

Assert that the compilation takes place in a freestanding environment

-ffunction-sections, -fno-function-sections

Place each function in its own section

-fgnu-inline-asm, -fno-gnu-inline-asm

-fgnu-keywords, -fno-gnu-keywords

Allow GNU-extension keywords regardless of language standard

-fgnu-runtime

Generate output compatible with the standard GNU Objective-C runtime

-fgnu89-inline, -fno-gnu89-inline

Use the gnu89 inline semantics

-fgnuc-version=<arg>

Sets various macros to claim compatibility with the given GCC version (default is 4.2.1)

-fgpu-allow-device-init, -fno-gpu-allow-device-init

Allow device side init function in HIP (experimental)

-fgpu-defer-diag, -fno-gpu-defer-diag

Defer host/device related diagnostic messages for CUDA/HIP

-fgpu-rdc, -fcuda-rdc, -fno-gpu-rdc

Generate relocatable device code, also known as separate compilation mode

-fgpu-sanitize, -fno-gpu-sanitize

Enable sanitizer for AMDGPU target

-fhip-fp32-correctly-rounded-divide-sqrt, -fno-hip-fp32-correctly-rounded-divide-sqrt

Specify that single precision floating-point divide and sqrt used in the program source are correctly rounded (HIP device compilation only)

-fhip-new-launch-api, -fno-hip-new-launch-api

Use new kernel launching API for HIP

-fhonor-infinities, -fhonor-infinites, -fno-honor-infinities

-fhonor-nans, -fno-honor-nans

-fhosted

-fignore-exceptions

Enable support for ignoring exception handling constructs

-fimplicit-module-maps, -fmodule-maps, -fno-implicit-module-maps

Implicitly search the file system for module map files.

-fimplicit-modules, -fno-implicit-modules

-finput-charset=<arg>

Specify the default character set for source files

-finstrument-function-entry-bare

Instrument function entry only, after inlining, without arguments to the instrumentation call

-finstrument-functions

Generate calls to instrument function entry and exit

-finstrument-functions-after-inlining

Like -finstrument-functions, but insert the calls after inlining

-fintegrated-as, -fno-integrated-as, -integrated-as

Enable the integrated assembler

-fintegrated-cc1, -fno-integrated-cc1

Run cc1 in-process

-fintegrated-objemitter, -fno-integrated-objemitter

Use internal machine object code emitter.

-fjmc, -fno-jmc

Enable just-my-code debugging

-fjump-tables, **-fno-jump-tables**

Use jump tables for lowering switches

-fkeep-static-consts, **-fno-keep-static-consts**

Keep static const variables if unused

-flax-vector-conversions=<arg>, **-flax-vector-conversions** (equivalent to -flax-vector-conversions=integer), **-fno-lax-vector-conversions** (equivalent to -flax-vector-conversions=none)

Enable implicit vector bit-casts. <arg> must be ‘none’, ‘integer’ or ‘all’.

-flimited-precision=<arg>

-flto-jobs=<arg>

Controls the backend parallelism of -flto=thin (default of 0 means the number of threads will be derived from the number of CPUs detected)

-flto=<arg>, **-flto** (equivalent to -flto=full), **-flto**=auto (equivalent to -flto=full), **-flto**=jobserver (equivalent to -flto=full)

Set LTO mode. <arg> must be ‘thin’ or ‘full’.

-fmacro-backtrace-limit=<arg>

-fmacro-prefix-map=<arg>

remap file source paths in predefined preprocessor macros and __builtin_FILE(). Implies -ffile-reproducible.

-fmargins=<arg1>,<arg2>...

Specifies, inclusively, the range of source column numbers that will be compiled

-fmath-errno, **-fno-math-errno**

Require math functions to indicate errors by setting errno

-fmax-tokens=<arg>

Max total number of preprocessed tokens for -Wmax-tokens.

-fmax-type-align=<arg>

Specify the maximum alignment to enforce on pointers lacking an explicit alignment

-fmemory-profile, **-fno-memory-profile**

Enable heap memory profiling

-fmemory-profile=<directory>

Enable heap memory profiling and dump results into <directory>

-fmerge-all-constants, **-fno-merge-all-constants**

Allow merging of constants

-fmessage-length=<arg>

Format message diagnostics so that they fit within N columns

-fminimize-whitespace, **-fno-minimize-whitespace**

Minimize whitespace when emitting preprocessor output

-fmodule-file-deps, **-fno-module-file-deps**

-fmodule-header

Build a C++20 Header Unit from a header.

-fmodule-header=<kind>

Build a C++20 Header Unit from a header that should be found in the user (fmodule-header=user) or system (fmodule-header=system) search path.

-fmodule-map-file=<file>

Load this module map file

-fmodule-name=<name>, **-fmodule-implementation-of** <arg>

Specify the name of the module to build

-fmodules, **-fno-modules**

Enable the 'modules' language feature

-fmodules-decluse, **-fno-modules-decluse**

Require declaration of modules used within a module

-fmodules-ignore-macro=<arg>

Ignore the definition of the given macro when building and loading modules

-fmodules-search-all, **-fno-modules-search-all**

Search even non-imported modules to resolve references

-fmodules-strict-decluse

Like -fmodules-decluse but requires all headers to be in modules

-fmodules-ts

Enable support for the C++ Modules TS

-fmodules-validate-input-files-content

Validate PCM input files based on content if mtime differs

-fms-compatibility, **-fno-ms-compatibility**

Enable full Microsoft Visual C++ compatibility

-fms-compatibility-version=<arg>

Dot-separated value representing the Microsoft compiler version number to report in _MSC_VER (0 = don't define it (default))

-fms-extensions, **-fno-ms-extensions**

Accept some non-standard constructs supported by the Microsoft compiler

-fms-hotpatch

Ensure that all functions can be hotpatched at runtime

-fms-memptr-rep=<arg>
<arg> must be 'single', 'multiple' or 'virtual'.

-fms-volatile

-fmsc-version=<arg>

Microsoft compiler version number to report in _MSC_VER (0 = don't define it (default))

-fmudflap

-fmudflapth

-fnested-functions

-fnew-alignment=<align>, **-fnew-alignment** <arg>

Specifies the largest alignment guaranteed by '::operator new(size_t)'

-fnew-infallible, **-fno-new-infallible**

Enable treating throwing global C++ operator new as always returning valid memory (annotates with __attribute__((returns_nonnull)) and throw()). This is detectable in source.

-fnext-runtime

-fno-builtin-<arg>

Disable implicit builtin knowledge of a specific function

-fno-elide-type

Do not elide types when printing diagnostics

-fno-knr-functions

Disable support for K&R C function declarations

-fno-margins

Specifies all range of source column numbers will be compiled

-fno-max-type-align

-fno-sequence

Specifies no columns are used for sequence numbers

-fno-strict-modules-decluse

-fno-temp-file

Directly create compilation output files. This may lead to incorrect incremental builds if the compiler crashes

-fno-working-directory

-fno_modules-validate-input-files-content

-fno_pch-validate-input-files-content

-fnoxray-link-deps

-fobjc-abi-version=<arg>

-fobjc-arc, **-fno-objc-arc**

Synthesize retain and release calls for Objective-C pointers

-fobjc-arc-exceptions, **-fno-objc-arc-exceptions**

Use EH-safe code when synthesizing retains and releases in -fobjc-arc

-fobjc-convert-messages-to-runtime-calls, **-fno-objc-convert-messages-to-runtime-calls**

-fobjc-disable-direct-methods-for-testing

Ignore attribute objc_direct so that direct methods can be tested

-fobjc-encode-cxx-class-template-spec, **-fno-objc-encode-cxx-class-template-spec**

Fully encode c++ class template specialization

-fobjc-exceptions, **-fno-objc-exceptions**

Enable Objective-C exceptions

-fobjc-infer-related-result-type, **-fno-objc-infer-related-result-type**

-fobjc-legacy-dispatch, **-fno-objc-legacy-dispatch**

-fobjc-link-runtime

-fobjc-nonfragile-abi, **-fno-objc-nonfragile-abi**

-fobjc-nonfragile-abi-version=<arg>

-fobjc-runtime=<arg>

Specify the target Objective-C runtime kind and version

-fobjc-sender-dependent-dispatch

-fobjc-weak, **-fno-objc-weak**

Enable ARC-style weak references in Objective-C

-foffload-lto=<arg>, **-foffload-lto** (equivalent to -foffload-lto=full)

Set LTO mode for offload compilation. <arg> must be ‘thin’ or ‘full’.

-fomit-frame-pointer, **-fno-omit-frame-pointer**

-fopenmp, **-fno-openmp**

Parse OpenMP pragmas and generate parallel code.

-fopenmp-extensions, **-fno-openmp-extensions**

Enable all Clang extensions for OpenMP directives and clauses

-fopenmp-implicit-rpath, **-fno-openmp-implicit-rpath**

Set rpath on OpenMP executables

-fopenmp-new-driver

Use the new driver for OpenMP offloading.

-fopenmp-offload-mandatory

Do not create a host fallback if offloading to the device fails.

-fopenmp-simd, **-fno-openmp-simd**

Emit OpenMP code only for SIMD-based constructs.

-fopenmp-target-debug, **-fno-openmp-target-debug**

Enable debugging in the OpenMP offloading device RTL

-fopenmp-version=<arg>

Set OpenMP version (e.g. 45 for OpenMP 4.5, 50 for OpenMP 5.0). Default value is 50.

-fopenmp=<arg>

-foperator-arrow-depth=<arg>

-foperator-names, **-fno-operator-names**

-foptimization-record-file=<file>

Specify the output name of the file containing the optimization remarks. Implies -fsave-optimization-record. On Darwin platforms, this cannot be used with multiple -arch <arch> options.

-foptimization-record-passes=<regex>

Only include passes which match a specified regular expression in the generated optimization record (by default, include all passes)

-foptimize-sibling-calls, **-fno-optimize-sibling-calls**

-forder-file-instrumentation

Generate instrumented code to collect order file into default.profraw file (overridden by ‘=’ form of option or LLVM_PROFILE_FILE env var)

-foutput-class-dir=<arg>, **--output-class-directory** <arg>,
--output-class-directory=<arg>

-fpack-struct, **-fno-pack-struct**

-fpack-struct=<arg>

Specify the default maximum struct packing alignment

-fpascal-strings, **-fno-pascal-strings**, **-mpascal-strings**

Recognize and construct Pascal-style string literals

-fpass-plugin=<dsopath>

Load pass plugin from a dynamic shared object file (only with new pass manager).

-fpatchable-function-entry=<N,M>

Generate M NOPs before function entry and N-M NOPs after function entry

-fpcc-struct-return

Override the default ABI to return all structs on the stack

-fpch-codegen, **-fno-pch-codegen**

Generate code for uses of this PCH that assumes an explicit object file will be built for the PCH

-fpch-debuginfo, **-fno-pch-debuginfo**

Generate debug info for types in an object file built from this PCH and do not generate them elsewhere

-fpch-instantiate-templates, **-fno-pch-instantiate-templates**

Instantiate templates already while building a PCH

-fpch-preprocess

-fpch-validate-input-files-content

Validate PCH input files based on content if mtime differs

-fpic, **-fno-pic**

-fpie, **-fno-pie**

-fplt, **-fno-plt**

-fplugin=<dsopath>

Load the named plugin (dynamic shared object)

-fprebuilt-implicit-modules, **-fno-prebuilt-implicit-modules**

Look up implicit modules in the prebuilt module path

-fpreserve-as-comments, **-fno-preserve-as-comments**

-fproc-stat-report<arg>

Print subprocess statistics

-fproc-stat-report=<arg>

Save subprocess statistics to the given file

-fprofile-arcs, **-fno-profile-arcs**

-fprofile-dir=<arg>

-fprofile-exclude-files=<arg>

Instrument only functions from files where names don’t match all the regexes separated by a semi-colon

-fprofile-filter-files=<arg>

Instrument only functions from files where names match any regex separated by a semi-colon

-fprofile-generate, **-fno-profile-generate**

Generate instrumented code to collect execution counts into default.profraw (overridden by LLVM_PROFILE_FILE env var)

-fprofile-generate=<directory>

Generate instrumented code to collect execution counts into <directory>/default.profraw (overridden by LLVM_PROFILE_FILE env var)

-fprofile-instr-generate, **-fno-profile-instr-generate**

Generate instrumented code to collect execution counts into default.profraw file (overridden by ‘=’ form of option or LLVM_PROFILE_FILE env var)

-fprofile-instr-generate=<file>

Generate instrumented code to collect execution counts into <file> (overridden by LLVM_PROFILE_FILE env var)

-fprofile-instr-use, **-fno-profile-instr-use**, **-fprofile-use**

-fprofile-instr-use=<arg>

Use instrumentation data for profile-guided optimization

-fprofile-list=<arg>

Filename defining the list of functions/files to instrument

-fprofile-remapping-file=<file>

Use the remappings described in <file> to match the profile data against names in the program

-fprofile-sample-accurate, -fauto-profile-accurate, -fno-profile-sample-accurate

Specifies that the sample profile is accurate. If the sample

profile is accurate, callsites without profile samples are marked as cold. Otherwise, treat callsites without profile samples as if we have no profile

-fprofile-sample-use, -fauto-profile, -fno-profile-sample-use

-fprofile-sample-use=<arg>, **-fauto-profile**=<arg>

Enable sample-based profile guided optimizations

-fprofile-update=<method>

Set update method of profile counters. <method> must be ‘atomic’, ‘prefer-atomic’ or ‘single’.

-fprofile-use=<pathname>

Use instrumentation data for profile-guided optimization. If pathname is a directory, it reads from <pathname>/default.profdata. Otherwise, it reads from file <pathname>.

-fprotect-parens, -fno-protect-parens

Determines whether the optimizer honors parentheses when floating-point expressions are evaluated

-fpseudo-probe-for-profiling, -fno-pseudo-probe-for-profiling

Emit pseudo probes for sample profiling

-freciprocal-math, -fno-reciprocal-math

Allow division operations to be reassociated

-freg-struct-return

Override the default ABI to return small structs in registers

-fregister-global-dtors-with-atexit, -fno-register-global-dtors-with-atexit

Use atexit or __cxa_atexit to register global destructors

-frelaxed-template-template-args, -fno-relaxed-template-template-args

Enable C++17 relaxed template template argument matching

-freroll-loops, -fno-reroll-loops

Turn on loop reroller

-frestrict-args, -fno-restrict-args

Assume all function parameters are restrict

-fretain-comments-from-system-headers

-frewrite-imports, -fno-rewrite-imports

-frewrite-includes, -fno-rewrite-includes

-frewrite-map-file=<arg>

-fropi, -fno-ropi

Generate read-only position independent code (ARM only)

-frounding-math, -fno-rounding-math

-frtti, **-fno-rtti**

-frtti-data, **-fno-rtti-data**

-frwpi, **-fno-rwpi**

Generate read-write position independent code (ARM only)

-fsanitize-memory-param-retval, **-fno-sanitize-memory-param-retval**

Enable detection of uninitialized parameters and return values

-fsave-optimization-record, **-fno-save-optimization-record**

Generate a YAML optimization record file

-fsave-optimization-record=<format>

Generate an optimization record file in a specific format

-fseh-exceptions

Use SEH style exceptions

-fsemantic-interposition, **-fno-semantic-interposition**

-fsequence=<arg1>,<arg2>...

Specifies the columns used for sequence numbers

-fshort-enums, **-fno-short-enums**

Allocate to an enum type only as many bytes as it needs for the declared range of possible values

-fshort-wchar, **-fno-short-wchar**

Force wchar_t to be a short unsigned int

-fshow-column, **-fno-show-column**

-fshow-overloads=<arg>

Which overload candidates to show when overload resolution fails. Defaults to ‘all’. <arg> must be ‘best’ or ‘all’.

-fshow-source-location, **-fno-show-source-location**

-fsignaling-math, **-fno-signaling-math**

-fsigned-bitfields

-fsigned-char, **-fno-signed-char**, **--signed-char**

char is signed

-fsigned-zeros, **-fno-signed-zeros**

-fsized-deallocation, **-fno-sized-deallocation**

Enable C++14 sized global deallocation functions

-fsjlj-exceptions

Use SjLj style exceptions

-fslmtags, **-fno-slmtags**

Enable IBM SLM tags logging

-fslp-vectorize, **-fno-slp-vectorize**, **-ftree-slp-vectorize**

Enable the superword-level parallelism vectorization passes

-fspell-checking, **-fno-spell-checking**

-fspell-checking-limit=<arg>

-fsplit-dwarf-inlining, **-fno-split-dwarf-inlining**

Provide minimal debug info in the object/executable to facilitate online symbolication/stack traces in the absence of .dwo/.dwp files when using Split DWARF

-fsplit-lto-unit, **-fno-split-lto-unit**

Enables splitting of the LTO unit

-fsplit-machine-functions, **-fno-split-machine-functions**

Enable late function splitting using profile information (x86 ELF)

-fsplit-stack, **-fno-split-stack**

Use segmented stack

-fstack-clash-protection, **-fno-stack-clash-protection**

Enable stack clash protection

-fstack-protector, **-fno-stack-protector**

Enable stack protectors for some functions vulnerable to stack smashing. This uses a loose heuristic which considers functions vulnerable if they contain a char (or 8bit integer) array or constant sized calls to alloca , which are of greater size than ssp-buffer-size (default: 8 bytes). All variable sized calls to alloca are considered vulnerable. A function with a stack protector has a guard value added to the stack frame that is checked on function exit. The guard value must be positioned in the stack frame such that a buffer overflow from a vulnerable variable will overwrite the guard value before overwriting the function's return address. The reference stack guard value is stored in a global variable.

-fstack-protector-all

Enable stack protectors for all functions

-fstack-protector-strong

Enable stack protectors for some functions vulnerable to stack smashing. Compared to -fstack-protector, this uses a stronger heuristic that includes functions containing arrays of any size (and any type), as well as any calls to alloca or the taking of an address from a local variable

-fstack-size-section, **-fno-stack-size-section**

Emit section containing metadata on function stack sizes

-fstack-usage

Emit .su file containing information on function stack sizes

-fstandalone-debug, **-fno-limit-debug-info**, **-fno-standalone-debug**

Emit full debug info for all types used by the program

-fstrict-aliasing, **-fno-strict-aliasing**

-fstrict-enums, **-fno-strict-enums**

Enable optimizations based on the strict definition of an enum's value range

-fstrict-float-cast-overflow, **-fno-strict-float-cast-overflow**

Assume that overflowing float-to-int casts are undefined (default)

-fstrict-overflow, **-fno-strict-overflow**

-fstrict-return, **-fno-strict-return**

-fstrict-vtable-pointers, **-fno-strict-vtable-pointers**

Enable optimizations based on the strict rules for overwriting polymorphic C++ objects

-fstruct-path-tbaa, **-fno-struct-path-tbaa**

-fswift-async-fp=<option>

Control emission of Swift async extended frame info. <option> must be 'auto', 'always' or 'never'.

-fsymbol-partition=<arg>

-ftabstop=<arg>

-ftemplate-backtrace-limit=<arg>

-ftemplate-depth-<arg>

-ftemplate-depth=<arg>

-ftest-coverage, **-fno-test-coverage**

-fthin-link-bitcode=<arg>

Write minimized bitcode to <file> for the ThinLTO thin link only

-fthinlto-index=<arg>

Perform ThinLTO importing using provided function summary index

-fthreadsafe-statics, **-fno-threadsafe-statics**

-ftime-report

-ftime-report=<arg>

(For new pass manager) 'per-pass': one report for each pass; 'per-pass-run': one report for each pass invocation. <arg> must be 'per-pass' or 'per-pass-run'.

-ftime-trace

Turn on time profiler. Generates JSON file based on output filename. Results can be analyzed with chrome://tracing or Speedscope App for flamegraph visualization.

-ftime-trace-granularity=<arg>

Minimum time granularity (in microseconds) traced by time profiler

-ftls-model=<arg>
<arg> must be 'global-dynamic', 'local-dynamic', 'initial-exec' or 'local-exec'.

-ftrap-function=<arg>

Issue call to specified function rather than a trap instruction

-ftrapping-math, **-fno-trapping-math**

-ftrapv

Trap on integer overflow

-ftrapv-handler <arg>

-ftrapv-handler=<function name>

Specify the function to be called on overflow

-ftrigraphs, **-fno-trigraphs**, **-trigraphs**, **--trigraphs**

Process trigraph sequences

-ftrivial-auto-var-init-stop-after=<arg>

Stop initializing trivial automatic stack variables after the specified number of instances

-ftrivial-auto-var-init=<arg>

Initialize trivial automatic stack variables. Defaults to 'uninitialized'. <arg> must be 'uninitialized', 'zero' or 'pattern'.

-funique-basic-block-section-names, **-fno-unique-basic-block-section-names**

Use unique names for basic block sections (ELF Only)

-funique-internal-linkage-names, **-fno-unique-internal-linkage-names**

Uniqueify Internal Linkage Symbol Names by appending the MD5 hash of the module path

-funique-section-names, **-fno-unique-section-names**

-funroll-loops, **-fno-unroll-loops**

Turn on loop unroller

-funsafe-math-optimizations, **-fno-unsafe-math-optimizations**

-funsigned-bitfields

-funsigned-char, **-fno-unsigned-char**, **--unsigned-char**

-funstable, **-fno-unstable**

Enable unstable and experimental features

-funwind-tables, **-fno-unwind-tables**

-fuse-cxa-atexit, **-fno-use-cxa-atexit**

-fuse-init-array, **-fno-use-init-array**

-fuse-ld=<arg>

-fuse-line-directives, **-fno-use-line-directives**

Use #line in preprocessed output

-fvalidate-ast-input-files-content

Compute and store the hash of input files used to build an AST. Files with mismatching mtime's are considered valid if both contents is identical

-fveclib=<arg>

Use the given vector functions library. <arg> must be 'Accelerate', 'libmvec', 'MASSV', 'SVML', 'Darwin_libsystem_m' or 'none'.

-fvectorize, **-fno-vectorize**, **-ftree-vectorize**

Enable the loop vectorization passes

-fverbose-asm, **-dA**, **-fno-verbose-asm**

Generate verbose assembly output

-fvirtual-function-elimination, **-fno-virtual-function-elimination**

Enables dead virtual function elimination optimization. Requires -flto=full

-fvisibility-dllexport=<arg>

The visibility for dllexport definitions [-fvisibility-from-dllstorageclass]. <arg> must be 'default', 'hidden', 'internal' or 'protected'.

-fvisibility-externs-dllimport=<arg>

The visibility for dllimport external declarations [-fvisibility-from-dllstorageclass]. <arg> must be 'default', 'hidden', 'internal' or 'protected'.

-fvisibility-externs-nodllstorageclass=<arg>

The visibility for external declarations without an explicit DLL dllstorageclass [-fvisibility-from-dllstorageclass]. <arg> must be 'default', 'hidden', 'internal' or 'protected'.

-fvisibility-from-dllstorageclass, **-fno-visibility-from-dllstorageclass**

Set the visibility of symbols in the generated code from their DLL storage class

-fvisibility-global-new-delete-hidden

Give global C++ operator new and delete declarations hidden visibility

-fvisibility-inlines-hidden, **-fno-visibility-inlines-hidden**

Give inline C++ member functions hidden visibility by default

-fvisibility-inlines-hidden-static-local-var,
-fno-visibility-inlines-hidden-static-local-var

When -fvisibility-inlines-hidden is enabled, static variables in inline C++ member functions will also be given hidden visibility by default

-fvisibility-ms-compat

Give global types 'default' visibility and global functions and variables 'hidden' visibility by default

-fvisibility-nodllstorageclass=<arg>

The visibility for definitions without an explicit DLL export class [-fvisibility-from-dllstorageclass]. <arg> must be ‘default’, ‘hidden’, ‘internal’ or ‘protected’.

-fvisibility=<arg>

Set the default symbol visibility for all global declarations. <arg> must be ‘hidden’ or ‘default’.

-fwasm-exceptions

Use WebAssembly style exceptions

-fwhole-program-vtables, -fno-whole-program-vtables

Enables whole-program vtable optimization. Requires -flto

-fwrapv, -fno-wrapv

Treat signed integer overflow as two’s complement

-fwritable-strings

Store string literals as writable data

-fxl-pragma-pack, -fno-xl-pragma-pack

Enable IBM XL #pragma pack handling

-fxray-always-emit-customevents, -fno-xray-always-emit-customevents

Always emit __xray_customevent(…) calls even if the containing function is not always instrumented

-fxray-always-emit-typedevents, -fno-xray-always-emit-typedevents

Always emit __xray_typedevent(…) calls even if the containing function is not always instrumented

-fxray-always-instrument=<arg>

DEPRECATED: Filename defining the whitelist for imbuing the ‘always instrument’ XRay attribute.

-fxray-attr-list=<arg>

Filename defining the list of functions/types for imbuing XRay attributes.

-fxray-function-groups=<arg>

Only instrument 1 of N groups

-fxray-function-index, -fno-xray-function-index

-fxray-ignore-loops, -fno-xray-ignore-loops

Don’t instrument functions with loops unless they also meet the minimum function size

-fxray-instruction-threshold<arg>

-fxray-instruction-threshold=<arg>

Sets the minimum function size to instrument with XRay

-fxray-instrument, -fno-xray-instrument

Generate XRay instrumentation sleds on function entry and exit

-fxray-instrumentation-bundle=<arg>

Select which XRay instrumentation points to emit. Options: all, none, function-entry, function-exit, function, custom. Default is ‘all’. ‘function’ includes both ‘function-entry’ and ‘function-exit’.

-fxray-link-deps

Tells clang to add the link dependencies for XRay.

-fxray-modes=<arg>

List of modes to link in by default into XRay instrumented binaries.

-fxray-never-instrument=<arg>

DEPRECATED: Filename defining the whitelist for imbuing the ‘never instrument’ XRay attribute.

-fxray-selected-function-group=<arg>

When using -fxray-function-groups, select which group of functions to instrument. Valid range is 0 to fxray-function-groups - 1

-fzero-call-used-regs=<arg>

Clear call-used registers upon function return (AArch64/x86 only). <arg> must be ‘skip’, ‘used-gpr-arg’, ‘used-gpr’, ‘used-arg’, ‘used’, ‘all-gpr-arg’, ‘all-gpr’, ‘all-arg’ or ‘all’.

-fzero-initialized-in-bss, **-fno-zero-initialized-in-bss**

-fzos-extensions, **-fno-zos-extensions**

Accept some non-standard constructs supported by the z/OS compiler

-fzos-le-char-mode=<mode>

Allow to select Language Environment character mode on z/OS to be <mode>

-fzvector, **-fno-zvector**, **-mzvector**

Enable System z vector language extension

--gpu-bundle-output, **--no-gpu-bundle-output**

Bundle output files of HIP device compilation

--offload-new-driver, **--no-offload-new-driver**

Use the new driver for offloading compilation.

-pedantic, **--pedantic**, **-no-pedantic**, **--no-pedantic**

Warn on language extensions

-pedantic-errors, **--pedantic-errors**

OpenCL flags

-cl-denorms-are-zero

OpenCL only. Allow denormals to be flushed to zero.

-cl-ext=<arg1>,<arg2>...

OpenCL only. Enable or disable OpenCL extensions/optional features. The argument is a comma-separated sequence of one or more extension names, each prefixed by ‘+’ or ‘-‘.

-cl-fast-relaxed-math

OpenCL only. Sets -cl-finite-math-only and -cl-unsafe-math-optimizations, and defines __FAST_RELAXED_MATH__.

-cl-finite-math-only

OpenCL only. Allow floating-point optimizations that assume arguments and results are not NaNs or +-Inf.

-cl-fp32-correctly-rounded-divide-sqrt

OpenCL only. Specify that single precision floating-point divide and sqrt used in the program source are correctly rounded.

-cl-kernel-arg-info

OpenCL only. Generate kernel argument metadata.

-cl-mad-enable

OpenCL only. Allow use of less precise MAD computations in the generated binary.

-cl-no-signed-zeros

OpenCL only. Allow use of less precise no signed zeros computations in the generated binary.

-cl-no-stdinc

OpenCL only. Disables all standard includes containing non-native compiler types and functions.

-cl-opt-disable

OpenCL only. This option disables all optimizations. By default optimizations are enabled.

-cl-single-precision-constant

OpenCL only. Treat double precision floating-point constant as single precision constant.

-cl-std=<arg>

OpenCL language standard to compile for. <arg> must be ‘cl’, ‘CL’, ‘cl1.0’, ‘CL1.0’, ‘cl1.1’, ‘CL1.1’, ‘cl1.2’, ‘CL1.2’, ‘cl2.0’, ‘CL2.0’, ‘cl3.0’, ‘CL3.0’, ‘clc++’, ‘CLC++’, ‘clc++1.0’, ‘CLC++1.0’, ‘clc++2021’ or ‘CLC++2021’.

-cl-strict-aliasing

OpenCL only. This option is added for compatibility with OpenCL 1.0.

-cl-uniform-work-group-size

OpenCL only. Defines that the global work-size be a multiple of the work-group size specified to clEnqueueNDRangeKernel

-cl-unsafe-math-optimizations

OpenCL only. Allow unsafe floating-point optimizations. Also implies -cl-no-signed-zeros and -cl-mad-enable.

SYCL flags

-fsycl, **-fno-sycl**

Enables SYCL kernels compilation for device

-sycl-std=<arg>

SYCL language standard to compile for. <arg> must be ‘2020’, ‘2017’, ‘121’, ‘1.2.1’ or ‘sycl-1.2.1’.

Target-dependent compilation options

-G<size>, **-G**=<arg>, **-msmall-data-limit**=<arg>, **-msmall-data-threshold**=<arg>

Put objects of at most <size> bytes into small data section (MIPS / Hexagon)

-ffixed-x1

Reserve the x1 register (AArch64/RISC-V only)

-ffixed-x10

Reserve the x10 register (AArch64/RISC-V only)

-ffixed-x11

Reserve the x11 register (AArch64/RISC-V only)

-ffixed-x12

Reserve the x12 register (AArch64/RISC-V only)

-ffixed-x13

Reserve the x13 register (AArch64/RISC-V only)

-ffixed-x14

Reserve the x14 register (AArch64/RISC-V only)

-ffixed-x15

Reserve the x15 register (AArch64/RISC-V only)

-ffixed-x16

Reserve the x16 register (AArch64/RISC-V only)

-ffixed-x17

Reserve the x17 register (AArch64/RISC-V only)

-ffixed-x18

Reserve the x18 register (AArch64/RISC-V only)

-ffixed-x19

Reserve the x19 register (AArch64/RISC-V only)

-ffixed-x2

Reserve the x2 register (AArch64/RISC-V only)

-ffixed-x20

Reserve the x20 register (AArch64/RISC-V only)

-ffixed-x21

Reserve the x21 register (AArch64/RISC-V only)

-ffixed-x22

Reserve the x22 register (AArch64/RISC-V only)

-ffixed-x23

Reserve the x23 register (AArch64/RISC-V only)

-ffixed-x24

Reserve the x24 register (AArch64/RISC-V only)

-ffixed-x25

Reserve the x25 register (AArch64/RISC-V only)

-ffixed-x26

Reserve the x26 register (AArch64/RISC-V only)

-ffixed-x27

Reserve the x27 register (AArch64/RISC-V only)

-ffixed-x28

Reserve the x28 register (AArch64/RISC-V only)

-ffixed-x29

Reserve the x29 register (AArch64/RISC-V only)

-ffixed-x3

Reserve the x3 register (AArch64/RISC-V only)

-ffixed-x30

Reserve the x30 register (AArch64/RISC-V only)

-ffixed-x31

Reserve the x31 register (AArch64/RISC-V only)

-ffixed-x4

Reserve the x4 register (AArch64/RISC-V only)

-ffixed-x5

Reserve the x5 register (AArch64/RISC-V only)

-ffixed-x6

Reserve the x6 register (AArch64/RISC-V only)

-ffixed-x7

Reserve the x7 register (AArch64/RISC-V only)

-ffixed-x8

Reserve the x8 register (AArch64/RISC-V only)

-ffixed-x9

Reserve the x9 register (AArch64/RISC-V only)

-ffuchsia-api-level=<arg>

Set Fuchsia API level

-inline-asm=<arg>
<arg> must be ‘att’ or ‘intel’.

-m16

-m32

-m64

-mabi=<arg>

-mabi=quadword-atomics

Enable quadword atomics ABI on AIX (AIX PPC64 only). Uses lqarx/stqcx. instructions.

-mabi=vec-default

Enable the default Altivec ABI on AIX (AIX only). Uses only volatile vector registers.

-mabi=vec-extabi

Enable the extended Altivec ABI on AIX (AIX only). Uses volatile and nonvolatile vector registers

-maix-struct-return

Return all structs in memory (PPC32 only)

-malign-branch-boundary=<arg>

Specify the boundary’s size to align branches

-malign-branch=<arg1>,<arg2>...

Specify types of branches to align

-malign-double

Align doubles to two words in structs (x86 only)

-mamdgpu-ieee, **-mno-amdgpu-ieee**

Sets the IEEE bit in the expected default floating point mode register. Floating point opcodes that support exception flag gathering quiet and propagate signaling NaN inputs per IEEE 754-2008. This option changes the ABI. (AMDGPU only)

-march=<arg>

-masm=<arg>

-mbackchain, **-mno-backchain**

Link stack frames through backchain on System Z

-mbranch-protection=<arg>

Enforce targets of indirect branches and function returns

-mbranches-within-32B-boundaries

Align selected branches (fused, jcc, jmp) within 32-byte boundary

-mcmodel=<arg>, **-mcmodel**=medany (equivalent to -mcmodel=medium), **-mcmodel**=medlow (equivalent to -mcmodel=small)

-mcode-object-v3, **-mno-code-object-v3**

Legacy option to specify code object ABI V3 (AMDGPU only)

-mcode-object-version=<arg>

Specify code object ABI version. Defaults to 4. (AMDGPU only). <arg> must be ‘none’, ‘2’, ‘3’, ‘4’ or ‘5’.

-mconsole<arg>

-mcpu=<arg>, **-mv5** (equivalent to -mcpu=hexagonv5), **-mv55** (equivalent to -mcpu=hexagonv55), **-mv60** (equivalent to -mcpu=hexagonv60), **-mv62** (equivalent to -mcpu=hexagonv62), **-mv65** (equivalent to -mcpu=hexagonv65), **-mv66** (equivalent to -mcpu=hexagonv66), **-mv67** (equivalent to -mcpu=hexagonv67), **-mv67t** (equivalent to -mcpu=hexagonv67t), **-mv68** (equivalent to -mcpu=hexagonv68), **-mv69** (equivalent to -mcpu=hexagonv69)

-mcrc, **-mno-crc**

Allow use of CRC instructions (ARM/Mips only)

-mcsect=<arg>, **-mcsect<arg>**

Set CSECT names in the output object module

-mdefault-build-attributes<arg>, **-mno-default-build-attributes<arg>**

-mdefault-visibility-export-mapping=<arg>

Mapping between default visibility and export. <arg> must be ‘none’, ‘explicit’ or ‘all’.

-mdll<arg>

-mdouble=<n

Force double to be <n> bits. <n must be ‘32’ or ‘64’.

-mdynamic-no-pic<arg>

-meabi <arg>

Set EABI type. Default depends on triple). <arg> must be ‘default’, ‘4’, ‘5’ or ‘gnu’.

-menable-experimental-extensions

Enable use of experimental RISC-V extensions.

-mfentry

Insert calls to fentry at function entry (x86/SystemZ only)

-mfloat-abi=<arg>
<arg> must be ‘soft’, ‘softfp’ or ‘hard’.

-mfpmath=<arg>

-mfpu=<arg>

-mgeneral-regs-only

Generate code which only uses the general purpose registers (AArch64/x86 only)

-mglobal-merge, **-mno-global-merge**

Enable merging of globals

-mhard-float

-mhwdiv=<arg>, **--mhwdiv** <arg>, **--mhwdiv**=<arg>

-mhwmult=<arg>

-miamcu, **-mno-iamcu**

Use Intel MCU ABI

-mibt-seal

Optimize fcf-protection=branch/full (requires LTO).

-mignore-xcoff-visibility

Not emit the visibility attribute for asm in AIX OS or give all symbols ‘unspecified’ visibility in XCOFF object file

-mimplicit-float, **-mno-implicit-float**

-mimplicit-it=<arg>

-mincremental-linker-compatible, **-mno-incremental-linker-compatible**

(integrated-as) Emit an object file which can be used with an incremental linker

-miphoneos-version-min=<arg>, **-mios-version-min**=<arg>

-mkernel

-mlong-calls, **-mno-long-calls**

Generate branches with extended addressability, usually via indirect jumps.

-mlvi-cfi, **-mno-lvi-cfi**

Enable only control-flow mitigations for Load Value Injection (LVI)

-mlvi-hardening, **-mno-lvi-hardening**

Enable all mitigations for Load Value Injection (LVI)

-mmacosx-version-min=<arg>, **-mmacos-version-min**=<arg>

Set Mac OS X deployment target

-mmcu=<arg>

-mms-bitfields, **-mno-ms-bitfields**

Set the default structure layout to be compatible with the Microsoft compiler standard

-mnocsect<arg>

Do not set CSECT names in the output object module

-mnop-mcount

Generate mcount/__fentry__ calls as nops. To activate they need to be patched in.

-momit-leaf-frame-pointer, **-mno-omit-leaf-frame-pointer**

Omit frame pointer setup for leaf functions

-moslib=<arg>

-mpacked-stack, **-mno-packed-stack**

Use packed stack layout (SystemZ only).

-mpad-max-prefix-size=<arg>

Specify maximum number of prefixes to use for padding

-mprefer-vector-width=<arg>

Specifies preferred vector width for auto-vectorization. Defaults to ‘none’ which allows target specific decisions.

-mqdsp6-compat

Enable hexagon-qdsp6 backward compatibility

-mrecip

-mrecip=<arg1>,<arg2>...

-mrecord-mcount

Generate a __mcount_loc section entry for each __fentry__ call.

-mred-zone, **-mno-red-zone**

-mregparm=<arg>

-mrelax, **-mno-relax**

Enable linker relaxation

-mrelax-all, **-mno-relax-all**

(integrated-as) Relax all machine instructions

-mretpoline, -mno-retpoline

-mrtd, -mno-rtd

Make StdCall calling convention the default

-mseses, -mno-seses

Enable speculative execution side effect suppression (SESES). Includes LVI control flow integrity mitigations

-msign-return-address=<arg>

Select return address signing scope. <arg> must be ‘none’, ‘all’ or ‘non-leaf’.

-msim

-mskip-rax-setup, -mno-skip-rax-setup

Skip setting up RAX register when passing variable arguments (x86 only)

-msoft-float, -mno-soft-float

Use software floating point

-mspeculative-load-hardening, -mno-speculative-load-hardening

-mstack-alignment=<arg>

Set the stack alignment

-mstack-arg-probe, -mno-stack-arg-probe

Enable stack probes

-mstack-probe-size=<arg>

Set the stack probe size

-mstack-protector-guard-offset=<arg>

Use the given offset for addressing the stack-protector guard

-mstack-protector-guard-reg=<arg>

Use the given reg for addressing the stack-protector guard

-mstack-protector-guard=<arg>

Use the given guard (global, tls) for addressing the stack-protector guard

-mstackrealign, -mno-stackrealign

Force realign the stack at entry to every function

-msvr4-struct-return

Return small structs in registers (PPC32 only)

-mtargetos=<arg>

Set the deployment target to be the specified OS and OS version

-mthread-model <arg>

The thread model to use. Defaults to ‘posix’). <arg> must be ‘posix’ or ‘single’.

-mthreads<arg>

-mthumb, -mno-thumb

-mtls-direct-seg-refs, -mno-tls-direct-seg-refs

Enable direct TLS access through segment registers (default)

-mtls-size=<arg>

Specify bit size of immediate TLS offsets (AArch64 ELF only): 12 (for 4KB) | 24 (for 16MB, default) | 32 (for 4GB) | 48 (for 256TB, needs -mcmodel=large)

-mtune=<arg>

Only supported on X86 and RISC-V. Otherwise accepted for compatibility with GCC.

-mtvos-version-min=<arg>, **-mappletvos-version-min**=<arg>

-municode<arg>

-munsafe-fp-atomics, **-mno-unsafe-fp-atomics**

Enable unsafe floating point atomic instructions (AMDGPU only)

-mvx, **-mno-vx**

-mwarn-nonportable-cfstrings, **-mno-warn-nonportable-cfstrings**

-mwatchos-version-min=<arg>

-mwavefrontsize64, **-mno-wavefrontsize64**

Specify wavefront size 64 mode (AMDGPU only)

-mwindows<arg>

-mx32

-mzos-target=<arg>

Set the z/OS release of the runtime environment

AARCH64

-fcall-saved-x10

Make the x10 register call-saved (AArch64 only)

-fcall-saved-x11

Make the x11 register call-saved (AArch64 only)

-fcall-saved-x12

Make the x12 register call-saved (AArch64 only)

-fcall-saved-x13

Make the x13 register call-saved (AArch64 only)

-fcall-saved-x14

Make the x14 register call-saved (AArch64 only)

-fcall-saved-x15

Make the x15 register call-saved (AArch64 only)

-fcall-saved-x18

Make the x18 register call-saved (AArch64 only)

-fcall-saved-x8

Make the x8 register call-saved (AArch64 only)

-fcall-saved-x9

Make the x9 register call-saved (AArch64 only)

-mfix-cortex-a53-835769, **-mno-fix-cortex-a53-835769**

Workaround Cortex-A53 erratum 835769 (AArch64 only)

-mmark-bti-property

Add .note.gnu.property with BTI to assembly files (AArch64 only)

-msve-vector-bits=<arg>

Specify the size in bits of an SVE vector register. Defaults to the vector length agnostic value of “scalable”. (AArch64 only)

-mvscale-max=<arg>

Specify the vscale maximum. Defaults to the vector length agnostic value of “0”. (AArch64 only)

-mvscale-min=<arg>

Specify the vscale minimum. Defaults to “1”. (AArch64 only)

AMDGPU

-mcumode, -mno-cumode

Specify CU wavefront execution mode (AMDGPU only)

-mtgsplit, -mno-tgsplit

Enable threadgroup split execution mode (AMDGPU only)

ARM

-faapcs-bitfield-load

Follows the AAPCS standard that all volatile bit-field write generates at least one load. (ARM only).

-faapcs-bitfield-width, -fno-aapcs-bitfield-width

Follow the AAPCS standard requirement stating that volatile bit-field width is dictated by the field container type. (ARM only).

-ffixed-r9

Reserve the r9 register (ARM only)

-mcmse

Allow use of CMSE (Armv8-M Security Extensions)

-mexecute-only, -mno-execute-only, -mpure-code

Disallow generation of data access to code sections (ARM only)

-mfix-cmse-cve-2021-35465, -mno-fix-cmse-cve-2021-35465

Work around VLLDM erratum CVE-2021-35465 (ARM only)

-mfix-cortex-a57-aes-1742098, -mfix-cortex-a72-aes-1655431, -mno-fix-cortex-a57-aes-1742098

Work around Cortex-A57 Erratum 1742098 (ARM only)

-mno-bti-at-return-twice

Do not add a BTI instruction after a setjmp or other return-twice construct (Arm/AArch64 only)

-mno-movt

Disallow use of movt/movw pairs (ARM only)

-mno-neg-immediates

Disallow converting instructions with negative immediates to their negation or inversion.

-mnocrc

Disallow use of CRC instructions (ARM only)

-mrestrict-it, -mno-restrict-it

Disallow generation of complex IT blocks.

-mtp=<arg>

Thread pointer access method (AArch32/AArch64 only). <arg> must be ‘soft’, ‘cp15’, ‘el0’, ‘el1’, ‘el2’ or ‘el3’.

-munaligned-access, -mno-unaligned-access

Allow memory accesses to be unaligned (AArch32/AArch64 only)

Hexagon

-mhvx-ieee-fp, -mno-hvx-ieee-fp

Enable Hexagon HVX IEEE floating-point

-mieee-rnd-near

-mmemops, -mno-memops

Enable generation of memop instructions

-mnvj, -mno-nvj

Enable generation of new-value jumps

-mnvs, -mno-nvs

Enable generation of new-value stores

-mpackets, -mno-packets

Enable generation of instruction packets

Hexagon

-mhvx, -mno-hvx

Enable Hexagon Vector eXtensions

-mhvx-length=<arg>

Set Hexagon Vector Length. <arg> must be ‘64B’ or ‘128B’.

-mhvx-qfloat, -mno-hvx-qfloat

Enable Hexagon HVX QFloat instructions

-mhvx=<arg>

Enable Hexagon Vector eXtensions

M68k

-ffixed-a0

Reserve the a0 register (M68k only)

-ffixed-a1

Reserve the a1 register (M68k only)

-ffixed-a2

Reserve the a2 register (M68k only)

-ffixed-a3

Reserve the a3 register (M68k only)

-ffixed-a4

Reserve the a4 register (M68k only)

-ffixed-a5

Reserve the a5 register (M68k only)

-ffixed-a6

Reserve the a6 register (M68k only)

-ffixed-d0

Reserve the d0 register (M68k only)

-ffixed-d1

Reserve the d1 register (M68k only)

-ffixed-d2

Reserve the d2 register (M68k only)

-ffixed-d3

Reserve the d3 register (M68k only)

-ffixed-d4

Reserve the d4 register (M68k only)

-ffixed-d5

Reserve the d5 register (M68k only)

-ffixed-d6

Reserve the d6 register (M68k only)

-ffixed-d7

Reserve the d7 register (M68k only)

-m68000

-m68010

-m68020

-m68030

-m68040

-m68060

MIPS

-mabicalls, -mno-abicalls

Enable SVR4-style position-independent code (Mips only)

-mabs=<arg>

-mcheck-zero-division, -mno-check-zero-division

-mcompact-branches=<arg>

-mdouble-float

-mdsp, -mno-dsp

-mdspr2, -mno-dspr2

-membedded-data, -mno-embedded-data

Place constants in the .rodata section instead of the .sdata section even if they meet the -G <size> threshold (MIPS)

-mextern-sdata, -mno-extern-sdata

Assume that externally defined data is in the small data if it meets the -G <size> threshold (MIPS)

-mfix4300

-mfp32

Use 32-bit floating point registers (MIPS only)

-mfp64

Use 64-bit floating point registers (MIPS only)

-mginv, **-mno-ginv**

-mgpopt, **-mno-gpopt**

Use GP relative accesses for symbols known to be in a small data section (MIPS)

-mindirect-jump=<arg>

Change indirect jump instructions to inhibit speculation

-mips16

-mldc1-sdc1, **-mno-ldc1-sdc1**

-mlocal-sdata, **-mno-local-sdata**

Extend the -G behaviour to object local data (MIPS)

-mmadd4, **-mno-madd4**

Enable the generation of 4-operand madd.s, madd.d and related instructions.

-mmicromips, **-mno-micromips**

-mmsa, **-mno-msa**

Enable MSA ASE (MIPS only)

-mmt, **-mno-mt**

Enable MT ASE (MIPS only)

-mnan=<arg>

-mno-mips16

-msingle-float

-mvirt, **-mno-virt**

-mxgot, **-mno-xgot**

PowerPC

-maltivec, **-mno-altivec**

-mcmpb, **-mno-cmpb**

-mcrbits, **-mno-crbits**

-mcrypto, **-mno-crypto**

-mdirect-move, **-mno-direct-move**

-mefpu2

-mfloat128, **-mno-float128**

-mfprnd, **-mno-fprnd**

-mhtm, **-mno-htm**

-minvariant-function-descriptors, **-mno-invariant-function-descriptors**

-misel, **-mno-isel**

-mlongcall, **-mno-longcall**

-mmfocrf, **-mmfcrf**, **-mno-mfocrf**

-mmma, **-mno-mma**

-mpaired-vector-memops, **-mno-paired-vector-memops**

-mpcrel, **-mno-pcrel**

-mpopcntd, **-mno-popcntd**

-mpower10-vector, **-mno-power10-vector**

-mpower8-vector, **-mno-power8-vector**

-mpower9-vector, **-mno-power9-vector**

-mprefixed, **-mno-prefixed**

-mprivileged

-mrop-protect

-msecure-plt

-mspe, **-mno-spe**

-mvsx, **-mno-vsx**

WebAssembly

-matomics, **-mno-atomics**

-mbulk-memory, **-mno-bulk-memory**

-mexception-handling, **-mno-exception-handling**

-mextended-const, **-mno-extended-const**

-mmultivalue, **-mno-multivalue**

-mmutable-globals, **-mno-mutable-globals**

-mnontrapping-fptoint, **-mno-nontrapping-fptoint**

-mreference-types, **-mno-reference-types**

-mrelaxed-simd, **-mno-relaxed-simd**

-msign-ext, **-mno-sign-ext**

-msimd128, **-mno-simd128**

-mtail-call, **-mno-tail-call**

WebAssembly Driver

-mexec-model=<arg>

Execution model (WebAssembly only). <arg> must be ‘command’ or ‘reactor’.

X86

-m3dnow, **-mno-3dnow**

-m3dnowa, **-mno-3dnowa**

-madx, **-mno-adx**

-maes, **-mno-aes**

-mamx-bf16, **-mno-amx-bf16**

-mamx-int8, **-mno-amx-int8**

-mamx-tile, **-mno-amx-tile**

-mavx, **-mno-avx**

-mavx2, **-mno-avx2**

-mavx512bf16, **-mno-avx512bf16**

-mavx512bitalg, **-mno-avx512bitalg**

-mavx512bw, **-mno-avx512bw**

-mavx512cd, **-mno-avx512cd**

-mavx512dq, **-mno-avx512dq**

-mavx512er, **-mno-avx512er**

-mavx512f, **-mno-avx512f**

-mavx512fp16, **-mno-avx512fp16**

-mavx512ifma, **-mno-avx512ifma**

-mavx512pf, **-mno-avx512pf**

-mavx512vbmi, **-mno-avx512vbmi**

-mavx512vbmi2, **-mno-avx512vbmi2**

-mavx512vl, **-mno-avx512vl**

-mavx512vnni, **-mno-avx512vnni**

-mavx512vp2intersect, **-mno-avx512vp2intersect**

-mavx512vpopcntdq, **-mno-avx512vpopcntdq**

-mavxvnni, **-mno-avxvnni**

-mbmi, **-mno-bmi**

-mbmi2, **-mno-bmi2**

-mcldemote, **-mno-cldemote**

-mclflushopt, **-mno-clflushopt**

-mclwb, **-mno-clwb**

-mclzero, **-mno-clzero**

-mcrc32, **-mno-crc32**

-mcx16, **-mno-cx16**

-menqcmd, **-mno-enqcmd**

-mf16c, **-mno-f16c**

-mfma, **-mno-fma**

-mfma4, **-mno-fma4**

-mfsgsbase, **-mno-fsgsbase**

-mfxsr, **-mno-fxsr**

-mgfni, **-mno-gfni**

-mhreset, **-mno-hreset**

-minvpcid, **-mno-invpcid**

-mkl, **-mno-kl**

-mlwp, **-mno-lwp**

-mlzcnt, **-mno-lzcnt**

-mmmx, **-mno-mmx**

-mmovbe, **-mno-movbe**

-mmovdir64b, **-mno-movdir64b**

-mmovdiri, **-mno-movdiri**

-mmwaitx, **-mno-mwaitx**

-mpclmul, **-mno-pclmul**

-mpconfig, **-mno-pconfig**

-mpku, **-mno-pku**

-mpopcnt, -mno-popcnt

-mprefetchwt1, -mno-prefetchwt1

-mprfchw, -mno-prfchw

-mptwrite, -mno-ptwrite

-mrdpid, -mno-rdpid

-mrdrnd, -mno-rdrnd

-mrdseed, -mno-rdseed

-mretpoline-external-thunk, -mno-retpoline-external-thunk

-mrtm, -mno-rtm

-msahf, -mno-sahf

-mserialize, -mno-serialize

-msgx, -mno-sgx

-msha, -mno-sha

-mshstk, -mno-shstk

-msse, -mno-sse

-msse2, -mno-sse2

-msse3, -mno-sse3

-msse4.1, -mno-sse4.1

-msse4.2, -mno-sse4.2, -msse4

-msse4a, -mno-sse4a

-mssse3, -mno-ssse3

-mtbm, -mno-tbm

-mtsxldtrk, -mno-tsxldtrk

-muintr, -mno-uintr

-mvaes, -mno-vaes

-mvpclmulqdq, -mno-vpclmulqdq

-mvzeroupper, -mno-vzeroupper

-mwaitpkg, -mno-waitpkg

-mwbnoinvd, -mno-wbnoinvd

-mwidekl, -mno-widekl

-mx87, -m80387, -mno-x87

-mxop, -mno-xop

-mxsave, -mno-xsave

-mxsavec, -mno-xsavec

-mxsaveopt, -mno-xsaveopt

-mxsaves, -mno-xsaves

RISCV

-msave-restore, -mno-save-restore

Enable using library calls for save and restore

Long double flags

Selects the long double implementation

-mlong-double-128

Force long double to be 128 bits

-mlong-double-64

Force long double to be 64 bits

-mlong-double-80

Force long double to be 80 bits, padded to 128 bits for storage

Optimization level

Flags controlling how much optimization should be performed.

-O<arg>, **-O** (equivalent to -O1), **--optimize**, **--optimize**=<arg>

-Ofast<arg>

Debug information generation

Flags controlling how much and what kind of debug information should be generated.

Kind and level of debug information

-g, **--debug**, **--debug**=<arg>

Generate source-level debug information

-gdwarf

Generate source-level debug information with the default dwarf version

-gdwarf-2

Generate source-level debug information with dwarf version 2

-gdwarf-3

Generate source-level debug information with dwarf version 3

-gdwarf-4

Generate source-level debug information with dwarf version 4

-gdwarf-5

Generate source-level debug information with dwarf version 5

-gdwarf32

Enables DWARF32 format for ELF binaries, if debug information emission is enabled.

-gdwarf64

Enables DWARF64 format for ELF binaries, if debug information emission is enabled.

-gfull

-ginline-line-tables, **-gno-inline-line-tables**

-gused

Debug level

-g0

-g2

-g3

-ggdb0

-ggdb1

-ggdb2

-ggdb3

-gline-directives-only

Emit debug line info directives only

-gline-tables-only, **-g1**, **-gmlt**

Emit debug line number tables only

-gmodules

Generate debug info with external references to clang modules or precompiled headers

Debugger to tune debug information for

-gdbx

-ggdb

-glldb

-gsce

Debug information flags

-gcolumn-info, **-gno-column-info**

-gdwarf-aranges

-gembed-source, **-gno-embed-source**

Embed source text in DWARF debug sections

-ggnu-pubnames, **-gno-gnu-pubnames**

-gpubnames, **-gno-pubnames**

-grecord-command-line, **-gno-record-command-line**, **-grecord-gcc-switches**

-gsimple-template-names, **-gno-simple-template-names**

-gsplit-dwarf, **-gno-split-dwarf**

-gsplit-dwarf=<arg>

Set DWARF fission mode. <arg> must be 'split' or 'single'.

-gstrict-dwarf, **-gno-strict-dwarf**

-gz=<arg>, **-gz** (equivalent to -gz=zlib)

DWARF debug sections compression type

Static analyzer flags

Flags controlling the behavior of the Clang Static Analyzer.

-Xanalyzer <arg>

Pass <arg> to the static analyzer

Fortran compilation flags

Flags that will be passed onto the `gfortran` compiler when Clang is given a Fortran input.

-A<arg>, **--assert** <arg>, **--assert**=<arg>

-A-<arg>

-faggressive-function-elimination, **-fno-aggressive-function-elimination**

-falign-commons, **-fno-align-commons**

-fall-intrinsics, **-fno-all-intrinsics**

-fbacktrace, **-fno-backtrace**

-fblas-matmul-limit=<arg>

-fbounds-check, **-fno-bounds-check**

-fcheck-array-temporaries, **-fno-check-array-temporaries**

-fcheck=<arg>

-fcoarray=<arg>

-fconvert=<arg>

-fcray-pointer, **-fno-cray-pointer**

-fd-lines-as-code, **-fno-d-lines-as-code**

-fd-lines-as-comments, **-fno-d-lines-as-comments**

-fdollar-ok, **-fno-dollar-ok**

-fdump-fortran-optimized, **-fno-dump-fortran-optimized**

-fdump-fortran-original, **-fno-dump-fortran-original**

-fdump-parse-tree, **-fno-dump-parse-tree**

-fexternal-blas, **-fno-external-blas**

-ff2c, **-fno-f2c**

-ffpe-trap=<arg>

-ffree-line-length-<arg>

-ffrontend-optimize, **-fno-frontend-optimize**

-finit-character=<arg>

-finit-integer=<arg>

-finit-local-zero, **-fno-init-local-zero**

-finit-logical=<arg>

-finit-real=<arg>

-finteger-4-integer-8, **-fno-integer-4-integer-8**

-fmax-array-constructor=<arg>

-fmax-errors=<arg>

-fmax-identifier-length, **-fno-max-identifier-length**

-fmax-stack-var-size=<arg>

-fmax-subrecord-length=<arg>

-fmodule-private, **-fno-module-private**

-fpack-derived, **-fno-pack-derived**

-frange-check, **-fno-range-check**

-freal-4-real-10, **-fno-real-4-real-10**

-freal-4-real-16, **-fno-real-4-real-16**

-freal-4-real-8, **-fno-real-4-real-8**

-freal-8-real-10, **-fno-real-8-real-10**

-freal-8-real-16, **-fno-real-8-real-16**

-freal-8-real-4, **-fno-real-8-real-4**

-frealloc-lhs, **-fno-realloc-lhs**

-frecord-marker=<arg>

-frecursive, **-fno-recursive**

-frepack-arrays, **-fno-repack-arrays**

-fsecond-underscore, **-fno-second-underscore**

-fsign-zero, **-fno-sign-zero**

-fstack-arrays, **-fno-stack-arrays**

-funderscoring, **-fno-underscoring**

-fwhole-file, **-fno-whole-file**

-imultilib <arg>

-static-libgfortran

Linker flags

Flags that are passed on to the linker

-L<dir>, **--library-directory** <arg>, **--library-directory**=<arg>

Add directory to library search path

-Mach

-T<script>

Specify <script> as linker script

-Tbss<addr>

Set starting address of BSS to <addr>

-Tdata<addr>

Set starting address of DATA to <addr>

-Ttext<addr>

Set starting address of TEXT to <addr>

-Wl,<arg>,<arg2>...

Pass the comma separated arguments in <arg> to the linker

-X

-Xlinker <arg>, **--for-linker** <arg>, **--for-linker**=<arg>

Pass <arg> to the linker

-Xoffload-linker<triple> <arg>

Pass <arg> to the offload linkers or the ones idenfied by -<triple>

-Z

-b<arg>

Pass -b <arg> to the linker on AIX (only).

-coverage, **--coverage**

-e<arg>, **--entry**

-filelist <arg>

--hip-device-lib=<arg>

HIP device library

--hipspv-pass-plugin=<dsopath>

path to a pass plugin for HIP to SPIR-V passes.

-l<arg>

--ld-path=<arg>

-nostartfiles

-nostdlib, **--no-standard-libraries**

--offload-link

Use the new offloading linker to perform the link job.

-pie

-r

-rdynamic

--rocm-device-lib-path=<arg>, **--hip-device-lib-path**=<arg>

ROCm device library path. Alternative to rocm-path.

-rpath <arg>

-s

-shared, **--shared**

-specs=<arg>, **--specs**=<arg>

-static, **--static**

-static-pie

-t

-u<arg>, **--force-link** <arg>, **--force-link**=<arg>

-undef

undef all system defines

-undefined<arg>, **--no-undefined**

-z <arg>

Pass -z <arg> to the linker

<clang-dxc options>

dxc compatibility options

/T<profile>, **-T<profile>**

Set target profile. <profile> must be ‘ps_6_0’, ‘ ps_6_1’, ‘ ps_6_2’, ‘ ps_6_3’, ‘ ps_6_4’, ‘ ps_6_5’, ‘ ps_6_6’, ‘ ps_6_7’, ‘vs_6_0’, ‘ vs_6_1’, ‘ vs_6_2’, ‘ vs_6_3’, ‘ vs_6_4’, ‘ vs_6_5’, ‘ vs_6_6’, ‘ vs_6_7’, ‘gs_6_0’, ‘ gs_6_1’, ‘ gs_6_2’, ‘ gs_6_3’, ‘ gs_6_4’, ‘ gs_6_5’, ‘ gs_6_6’, ‘ gs_6_7’, ‘hs_6_0’, ‘ hs_6_1’, ‘ hs_6_2’, ‘ hs_6_3’, ‘ hs_6_4’, ‘ hs_6_5’, ‘ hs_6_6’, ‘ hs_6_7’, ‘ds_6_0’, ‘ ds_6_1’, ‘ ds_6_2’, ‘ ds_6_3’, ‘ ds_6_4’, ‘ ds_6_5’, ‘ ds_6_6’, ‘ ds_6_7’, ‘cs_6_0’, ‘ cs_6_1’, ‘ cs_6_2’, ‘ cs_6_3’, ‘ cs_6_4’, ‘ cs_6_5’, ‘ cs_6_6’, ‘ cs_6_7’, ‘lib_6_3’, ‘ lib_6_4’, ‘ lib_6_5’, ‘ lib_6_6’, ‘ lib_6_7’, ‘ lib_6_x’, ‘ms_6_5’, ‘ ms_6_6’, ‘ ms_6_7’, ‘as_6_5’, ‘ as_6_6’ or ‘ as_6_7’.

/emit-pristine-llvm, **-emit-pristine-llvm**, **/fcgl**, **-fcgl**

Emit pristine LLVM IR from the frontend by not running any LLVM passes at all.Same as -S + -emit-llvm + -disable-llvm-passes.

/hlsl-no-stdinc, **-hlsl-no-stdinc**

HLSL only. Disables all standard includes containing non-native compiler types and functions.

Attributes in Clang

Introduction 244
AMD GPU Attributes 245
amdgpu_flat_work_group_size 245
amdgpu_num_sgpr 245
amdgpu_num_vgpr 246
amdgpu_waves_per_eu 246
Calling Conventions 247
aarch64_sve_pcs 247
aarch64_vector_pcs 247
fastcall 248
ms_abi 248
pcs 248
preserve_all 249
preserve_most 249
regcall 250
regparm 250
stdcall 250
thiscall 250
vectorcall 251
Consumed Annotation Checking 251
callable_when 251
consumable 252
param_typestate 252
return_typestate 252
set_typestate 252
test_typestate 253
Customizing Swift Import 253
swift_async 253
swift_async_error 254
swift_async_name 254
swift_attr 255
swift_bridge 255
swift_bridged 255
swift_error 256
swift_name 256
swift_newtype 257
swift_objc_members 257
swift_private 257
Declaration Attributes 258
Owner 258
Pointer 258
_Packed 259
__single_inhertiance, __multiple_inheritance, __virtual_inheritance 260

asm 260
deprecated 261
empty_bases 261
enum_extensibility 262
external_source_symbol 263
flag_enum 263
layout_version 264
lto_visibility_public 264
managed 264
novtable 264
ns_error_domain 265
objc_boxable 265
objc_direct 266
objc_direct_members 267
objc_non_runtime_protocol 267
objc_nonlazy_class 268
objc_runtime_name 268
objc_runtime_visible 268
objc_subclassing_restricted 269
preferred_name 269
randomize_layout, no_randomize_layout 269
randomize_layout, no_randomize_layout 270
selectany 270
transparent_union 270
trivial_abi 271
using_if_exists 272
Field Attributes 272
no_unique_address 272
Function Attributes 273
#pragma omp declare simd 273
#pragma omp declare target 273
#pragma omp declare variant 274
SV_GroupIndex 275
_Export 275
_Noreturn 275
abi_tag 275
acquire_capability, acquire_shared_capability 276
alloc_align 276
alloc_size 277
allocator 277
always_inline, __force_inline 278
artificial 278
assert_capability, assert_shared_capability 279
assume 279
assume_aligned 279

availability 280
btf_decl_tag 282
callback 282
carries_dependency 283
cf_consumed 283
cf_returns_not_retained 284
cf_returns_retained 285
cfi_canonical_jump_table 286
clang::builtin_alias, clang_builtin_alias 286
clang_arm_builtin_alias 287
cmse_nonsecure_entry 287
code_seg 287
convergent 288
cpu_dispatch 288
cpu_specific 289
diagnose_as_builtin 290
diagnose_if 291
disable_sanitizer_instrumentation 292
disable_tail_calls 292
enable_if 293
enforce_tcb 295
enforce_tcb_leaf 295
error, warning 295
exclude_from_explicit_instantiation 296
export_name 297
flatten 297
force_align_arg_pointer 297
format 298
gnu_inline 299
guard 299
ifunc 300
import_module 300
import_name 300
internal_linkage 301
interrupt (ARM) 301
interrupt (AVR) 302
interrupt (MIPS) 302
interrupt (RISCV) 302
kernel 303
lifetimebound 303
long_call, far 304
malloc 304
micromips 305
mig_server_routine 305
min_vector_width 305

no_builtin 306
no_caller_saved_registers 306
no_profile_instrument_function 307
no_sanitize 307
no_sanitize_address, no_address_safety_analysis 308
no_sanitize_memory 308
no_sanitize_thread 308
no_speculative_load_hardening 309
no_split_stack 310
no_stack_protector 310
noalias 310
nocf_check 310
nodiscard, warn_unused_result 311
noduplicate 311
noinline 312
nomicromips 312
noreturn, _Noreturn 313
not_tail_called 313
nothrow 314
ns_consumed 314
ns_consumes_self 315
ns_returns_autoreleased 316
ns_returns_not_retained 317
ns_returns_retained 317
numthreads 318
objc_method_family 319
objc_requires_super 319
optnone 320
os_consumed 320
os_consumes_this 321
os_returns_not_retained 322
os_returns_retained 322
os_returns_retained_on_non_zero 323
os_returns_retained_on_zero 324
overloadable 325
patchable_function_entry 326
preserve_access_index 327
reinitializes 327
release_capability, release_shared_capability 328
retain 328
shader 328
short_call, near 329
signal 329
speculative_load_hardening 329
sycl_kernel 330

target 331
target_clones 332
try_acquire_capability, try_acquire_shared_capability 332
used 333
xray_always_instrument, xray_never_instrument, xray_log_args 333
xray_always_instrument, xray_never_instrument, xray_log_args 334
zero_call_used_regs 334
Handle Attributes 335
acquire_handle 335
release_handle 335
use_handle 335
Nullability Attributes 336
_Nonnull 336
_Null_unspecified 337
_Nullable 337
_Nullable_result 337
nonnull 338
returns_nonnull 338
OpenCL Address Spaces 339
[[clang::opencl_global_device]], [[clang::opencl_global_host]] 339
[[clang::opencl_global_device]], [[clang::opencl_global_host]] 339
__constant, constant, [[clang::opencl_constant]] 340
__generic, generic, [[clang::opencl_generic]] 340
__global, global, [[clang::opencl_global]] 340
__local, local, [[clang::opencl_local]] 341
__private, private, [[clang::opencl_private]] 341
Statement Attributes 341
#pragma clang loop 341
#pragma unroll, #pragma nounroll 342
__read_only, __write_only, __read_write (read_only, write_only, read_write) 343
fallthrough 344
intel_reqd_sub_group_size 345
likely and unlikely 345
likely and unlikely 347
musttail 350
nomerge 350
opencl_unroll_hint 351
suppress 351
sycl_special_class 351
Type Attributes 352
__ptr32 352
__ptr64 353
__sptr 353
__uptr 353
align_value 353

arm_sve_vector_bits 354
btf_type_tag 354
clang_arm_mve_strict_polymorphism 354
cmse_nonsecure_call 355
device_builtin_surface_type 355
device_builtin_texture_type 356
noderef 356
objc_class_stub 357
Type Safety Checking **357**
argument_with_type_tag 358
pointer_with_type_tag 358
type_tag_for_datatype 359
Variable Attributes **361**
always_destroy 361
called_once 361
dllexport 362
dllimport 362
init_priority 363
init_seg 363
leaf 363
loader_uninitialized 364
maybe_unused, unused 364
no_destroy 364
nodebug 365
noescape 365
nosvm 366
objc_externally_retained 366
pass_object_size, pass_dynamic_object_size 367
require_constant_initialization, constinit (C++20) 369
section, __declspec(allocate) 369
standalone_debug 370
swift_async_context 370
swift_context 370
swift_error_result 370
swift_indirect_result 371
swiftasynccall 372
swiftcall 372
thread 373
tls_model 373
uninitialized 373

Introduction

This page lists the attributes currently supported by Clang.

AMD GPU Attributes

amdgpu_flat_work_group_size

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
amdgpu_flat_work_group_size	clang::amdgpu_flat_work_group_size						Yes

The flat work-group size is the number of work-items in the work-group size specified when the kernel is dispatched. It is the product of the sizes of the x, y, and z dimension of the work-group.

Clang supports the `__attribute__((amdgpu_flat_work_group_size(<min>, <max>)))` attribute for the AMDGPU target. This attribute may be attached to a kernel function definition and is an optimization hint.

`<min>` parameter specifies the minimum flat work-group size, and `<max>` parameter specifies the maximum flat work-group size (must be greater than `<min>`) to which all dispatches of the kernel will conform. Passing `0, 0` as `<min>, <max>` implies the default behavior (`128, 256`).

If specified, the AMDGPU target backend might be able to produce better machine code for barriers and perform scratch promotion by estimating available group segment size.

An error will be given if:

- Specified values violate subtarget specifications;
- Specified values are not compatible with values provided through other attributes.

amdgpu_num_sgpr

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
amdgpu_num_sgpr	clang::amdgpu_num_sgpr						Yes

Clang supports the `__attribute__((amdgpu_num_sgpr(<num_sgpr>)))` and `__attribute__((amdgpu_num_vgpr(<num_vgpr>)))` attributes for the AMDGPU target. These attributes may be attached to a kernel function definition and are an optimization hint.

If these attributes are specified, then the AMDGPU target backend will attempt to limit the number of SGPRs and/or VGPRs used to the specified value(s). The number of used SGPRs and/or VGPRs may further be rounded up to satisfy the allocation requirements or constraints of the subtarget. Passing `0` as `num_sgpr` and/or `num_vgpr` implies the default behavior (no limits).

These attributes can be used to test the AMDGPU target backend. It is recommended that the `amdgpu_waves_per_eu` attribute be used to control resources such as SGPRs and VGPRs since it is aware of the limits for different subtargets.

An error will be given if:

- Specified values violate subtarget specifications;

- Specified values are not compatible with values provided through other attributes;
- The AMDGPU target backend is unable to create machine code that can meet the request.

amdgpu_num_vgpr

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
amdgpu_num_vgpr	clang::amdgpu_num_vgpr						Yes

Clang supports the `__attribute__((amdgpu_num_sgpr(<num_sgpr>)))` and `__attribute__((amdgpu_num_vgpr(<num_vgpr>)))` attributes for the AMDGPU target. These attributes may be attached to a kernel function definition and are an optimization hint.

If these attributes are specified, then the AMDGPU target backend will attempt to limit the number of SGPRs and/or VGPRs used to the specified value(s). The number of used SGPRs and/or VGPRs may further be rounded up to satisfy the allocation requirements or constraints of the subtarget. Passing `0` as `num_sgpr` and/or `num_vgpr` implies the default behavior (no limits).

These attributes can be used to test the AMDGPU target backend. It is recommended that the `amdgpu_waves_per_eu` attribute be used to control resources such as SGPRs and VGPRs since it is aware of the limits for different subtargets.

An error will be given if:

- Specified values violate subtarget specifications;
- Specified values are not compatible with values provided through other attributes;
- The AMDGPU target backend is unable to create machine code that can meet the request.

amdgpu_waves_per_eu

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
amdgpu_waves_per_eu	clang::amdgpu_waves_per_eu						Yes

A compute unit (CU) is responsible for executing the wavefronts of a work-group. It is composed of one or more execution units (EU), which are responsible for executing the wavefronts. An EU can have enough resources to maintain the state of more than one executing wavefront. This allows an EU to hide latency by switching between wavefronts in a similar way to symmetric multithreading on a CPU. In order to allow the state for multiple wavefronts to fit on an EU, the resources used by a single wavefront have to be limited. For example, the number of SGPRs and VGPRs. Limiting such resources can allow greater latency hiding, but can result in having to spill some register state to memory.

Clang supports the `__attribute__((amdgpu_waves_per_eu(<min>[, <max>])))` attribute for the AMDGPU target. This attribute may be attached to a kernel function definition and is an optimization hint.

`<min>` parameter specifies the requested minimum number of waves per EU, and *optional* `<max>` parameter specifies the requested maximum number of waves per EU (must be greater than `<min>` if specified). If `<max>` is

omitted, then there is no restriction on the maximum number of waves per EU other than the one dictated by the hardware for which the kernel is compiled. Passing `0, 0` as `<min>, <max>` implies the default behavior (no limits).

If specified, this attribute allows an advanced developer to tune the number of wavefronts that are capable of fitting within the resources of an EU. The AMDGPU target backend can use this information to limit resources, such as number of SGPRs, number of VGPRs, size of available group and private memory segments, in such a way that guarantees that at least `<min>` wavefronts and at most `<max>` wavefronts are able to fit within the resources of an EU. Requesting more wavefronts can hide memory latency but limits available registers which can result in spilling. Requesting fewer wavefronts can help reduce cache thrashing, but can reduce memory latency hiding.

This attribute controls the machine code generated by the AMDGPU target backend to ensure it is capable of meeting the requested values. However, when the kernel is executed, there may be other reasons that prevent meeting the request, for example, there may be wavefronts from other kernels executing on the EU.

An error will be given if:

- Specified values violate subtarget specifications;
- Specified values are not compatible with values provided through other attributes;
- The AMDGPU target backend is unable to create machine code that can meet the request.

Calling Conventions

Clang supports several different calling conventions, depending on the target platform and architecture. The calling convention used for a function determines how parameters are passed, how results are returned to the caller, and other low-level details of calling a function.

aarch64_sve_pcs

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
aarch64_sve_pcs	clang::aarch64_sve_pcs	clang::aarch64_sve_pcs					

On AArch64 targets, this attribute changes the calling convention of a function to preserve additional Scalable Vector registers and Scalable Predicate registers relative to the default calling convention used for AArch64.

This means it is more efficient to call such functions from code that performs extensive scalable vector and scalable predicate calculations, because fewer live SVE registers need to be saved. This property makes it well-suited for SVE math library functions, which are typically leaf functions that require a small number of registers.

However, using this attribute also means that it is more expensive to call a function that adheres to the default calling convention from within such a function. Therefore, it is recommended that this attribute is only used for leaf functions.

For more information, see the documentation for *aarch64_sve_pcs* in the ARM C Language Extension (ACLE) documentation.

aarch64_vector_pcs

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
aarch64_vector_pcs	clang::aarch64_vector_pcs	clang::aarch64_vector_pcs					

On AArch64 targets, this attribute changes the calling convention of a function to preserve additional floating-point and Advanced SIMD registers relative to the default calling convention used for AArch64.

This means it is more efficient to call such functions from code that performs extensive floating-point and vector calculations, because fewer live SIMD and FP registers need to be saved. This property makes it well-suited for e.g. floating-point or vector math library functions, which are typically leaf functions that require a small number of registers.

However, using this attribute also means that it is more expensive to call a function that adheres to the default calling convention from within such a function. Therefore, it is recommended that this attribute is only used for leaf functions.

For more information, see the documentation for aarch64_vector_pcs on the Arm Developer website.

fastcall

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
fastcall	gnu::fastcall	gnu::fastcall		__fastcall _fastcall			

On 32-bit x86 targets, this attribute changes the calling convention of a function to use ECX and EDX as register parameters and clear parameters off of the stack on return. This convention does not support variadic calls or unprototyped functions in C, and has no effect on x86_64 targets. This calling convention is supported primarily for compatibility with existing code. Users seeking register parameters should use the `regparm` attribute, which does not require callee-cleanup. See the documentation for __fastcall on MSDN.

ms_abi

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
ms_abi	gnu::ms_abi	gnu::ms_abi					

On non-Windows x86_64 targets, this attribute changes the calling convention of a function to match the default convention used on Windows x86_64. This attribute has no effect on Windows targets or non-x86_64 targets.

pcs

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
pcs	gnu::pcs	gnu::pcs					

On ARM targets, this attribute can be used to select calling conventions similar to `stdcall` on x86. Valid parameter values are “aapcs” and “aapcs-vfp”.

preserve_all

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
preserve_all	clang::preserve_all	clang::preserve_all					

On X86-64 and AArch64 targets, this attribute changes the calling convention of a function. The `preserve_all` calling convention attempts to make the code in the caller even less intrusive than the `preserve_most` calling convention. This calling convention also behaves identical to the `C` calling convention on how arguments and return values are passed, but it uses a different set of caller/callee-saved registers. This removes the burden of saving and recovering a large register set before and after the call in the caller. If the arguments are passed in callee-saved registers, then they will be preserved by the callee across the call. This doesn’t apply for values returned in callee-saved registers.

- On X86-64 the callee preserves all general purpose registers, except for R11. R11 can be used as a scratch register. Furthermore it also preserves all floating-point registers (XMMs/YMMs).

The idea behind this convention is to support calls to runtime functions that don’t need to call out to any other functions.

This calling convention, like the `preserve_most` calling convention, will be used by a future version of the Objective-C runtime and should be considered experimental at this time.

preserve_most

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
preserve_most	clang::preserve_most	clang::preserve_most					

On X86-64 and AArch64 targets, this attribute changes the calling convention of a function. The `preserve_most` calling convention attempts to make the code in the caller as unintrusive as possible. This convention behaves identically to the `C` calling convention on how arguments and return values are passed, but it uses a different set of caller/callee-saved registers. This alleviates the burden of saving and recovering a large register set before and after the call in the caller. If the arguments are passed in callee-saved registers, then they will be preserved by the callee across the call. This doesn’t apply for values returned in callee-saved registers.

- On X86-64 the callee preserves all general purpose registers, except for R11. R11 can be used as a scratch register. Floating-point registers (XMMs/YMMs) are not preserved and need to be saved by the caller.

The idea behind this convention is to support calls to runtime functions that have a hot path and a cold path. The hot path is usually a small piece of code that doesn’t use many registers. The cold path might need to call out to another function and therefore only needs to preserve the caller-saved registers, which haven’t already been saved by the

caller. The `preserve_most` calling convention is very similar to the `cold` calling convention in terms of caller/callee-saved registers, but they are used for different types of function calls. `coldcc` is for function calls that are rarely executed, whereas `preserve_most` function calls are intended to be on the hot path and definitely executed a lot. Furthermore `preserve_most` doesn't prevent the inliner from inlining the function call.

This calling convention will be used by a future version of the Objective-C runtime and should therefore still be considered experimental at this time. Although this convention was created to optimize certain runtime calls to the Objective-C runtime, it is not limited to this runtime and might be used by other runtimes in the future too. The current implementation only supports X86-64 and AArch64, but the intention is to support more architectures in the future.

regcall

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
regcall	gnu::regcall	gnu::regcall		__regcall			

On x86 targets, this attribute changes the calling convention to __regcall convention. This convention aims to pass as many arguments as possible in registers. It also tries to utilize registers for the return value whenever it is possible.

regparm

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
regparm	gnu::regparm	gnu::regparm					

On 32-bit x86 targets, the regparm attribute causes the compiler to pass the first three integer parameters in EAX, EDX, and ECX instead of on the stack. This attribute has no effect on variadic functions, and all parameters are passed via the stack as normal.

stdcall

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
stdcall	gnu::stdcall	gnu::stdcall		__stdcall _stdcall			

On 32-bit x86 targets, this attribute changes the calling convention of a function to clear parameters off of the stack on return. This convention does not support variadic calls or unprototyped functions in C, and has no effect on x86_64 targets. This calling convention is used widely by the Windows API and COM applications. See the documentation for __stdcall on MSDN.

thiscall

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
thiscall	gnu::thiscall	gnu::thiscall		__thiscall _thiscall			

On 32-bit x86 targets, this attribute changes the calling convention of a function to use ECX for the first parameter (typically the implicit `this` parameter of C++ methods) and clear parameters off of the stack on return. This convention does not support variadic calls or unprototyped functions in C, and has no effect on x86_64 targets. See the documentation for __thiscall on MSDN.

vectorcall

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
vectorcall	clang::vectorcall	clang::vectorcall		__vectorcall _vectorcall			

On 32-bit x86 *and* x86_64 targets, this attribute changes the calling convention of a function to pass vector parameters in SSE registers.

On 32-bit x86 targets, this calling convention is similar to `__fastcall`. The first two integer parameters are passed in ECX and EDX. Subsequent integer parameters are passed in memory, and callee clears the stack. On x86_64 targets, the callee does *not* clear the stack, and integer parameters are passed in RCX, RDX, R8, and R9 as is done for the default Windows x64 calling convention.

On both 32-bit x86 and x86_64 targets, vector and floating point arguments are passed in XMM0-XMM5. Homogeneous vector aggregates of up to four elements are passed in sequential SSE registers if enough are available. If AVX is enabled, 256 bit vectors are passed in YMM0-YMM5. Any vector or aggregate type that cannot be passed in registers for any reason is passed by reference, which allows the caller to align the parameter memory.

See the documentation for __vectorcall on MSDN for more details.

Consumed Annotation Checking

Clang supports additional attributes for checking basic resource management properties, specifically for unique objects that have a single owning reference. The following attributes are currently supported, although **the implementation for these annotations is currently in development and are subject to change.**

callable_when

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
callable_when	clang::callable_when						Yes

Use `__attribute__((callable_when(...)))` to indicate what states a method may be called in. Valid states are unconsumed, consumed, or unknown. Each argument to this attribute must be a quoted string. E.g.:

```
__attribute__((callable_when("unconsumed", "unknown")))
```

consumable

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
consumable	clang::consumable						Yes

Each `class` that uses any of the typestate annotations must first be marked using the `consumable` attribute. Failure to do so will result in a warning.

This attribute accepts a single parameter that must be one of the following: `unknown`, `consumed`, or `unconsumed`.

param_typestate

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
param_typestate	clang::param_typestate						Yes

This attribute specifies expectations about function parameters. Calls to an function with annotated parameters will issue a warning if the corresponding argument isn't in the expected state. The attribute is also used to set the initial state of the parameter when analyzing the function's body.

return_typestate

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
return_typestate	clang::return_typestate						Yes

The `return_typestate` attribute can be applied to functions or parameters. When applied to a function the attribute specifies the state of the returned value. The function's body is checked to ensure that it always returns a value in the specified state. On the caller side, values returned by the annotated function are initialized to the given state.

When applied to a function parameter it modifies the state of an argument after a call to the function returns. The function's body is checked to ensure that the parameter is in the expected state before returning.

set_typestate

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
set_typestate	clang::set_typestate						Yes

Annotate methods that transition an object into a new state with `__attribute__((set_typestate(new_state)))`. The new state must be unconsumed, consumed, or unknown.

test_typestate

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
test_typestate	clang::test_typestate						Yes

Use `__attribute__((test_typestate(tested_state)))` to indicate that a method returns true if the object is in the specified state..

Customizing Swift Import

Clang supports additional attributes for customizing how APIs are imported into Swift.

swift_async

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
swift_async	clang::swift_async	clang::swift_async					Yes

The `swift_async` attribute specifies if and how a particular function or Objective-C method is imported into a swift async method. For instance:

```
@interface MyClass : NSObject
-(void)notActuallyAsync:(int)p1 withCompletionHandler:(void (^)())handler
    __attribute__((swift_async(none)));

-(void)actuallyAsync:(int)p1 callThisAsync:(void (^)())fun
    __attribute__((swift_async(swift_private, 1)));
@end
```

Here, `notActuallyAsync:withCompletionHandler` would have been imported as `async` (because it's last parameter's selector piece is `withCompletionHandler`) if not for the `swift_async(none)` attribute. Conversely, `actuallyAsync:callThisAsync` wouldn't have been imported as `async` if not for the `swift_async` attribute because it doesn't match the naming convention.

When using `swift_async` to enable importing, the first argument to the attribute is either `swift_private` or `not_swift_private` to indicate whether the function/method is private to the current framework, and the second argument is the index of the completion handler parameter.

swift_async_error

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
swift_async_error	clang::swift_async_error	clang::swift_async_error					Yes

The `swift_async_error` attribute specifies how an error state will be represented in a swift async method. It's a bit analogous to the `swift_error` attribute for the generated async method. The `swift_async_error` attribute can indicate a variety of different ways of representing an error.

- `__attribute__((swift_async_error(zero_argument, N)))`, specifies that the async method is considered to have failed if the Nth argument to the completion handler is zero.
- `__attribute__((swift_async_error(nonzero_argument, N)))`, specifies that the async method is considered to have failed if the Nth argument to the completion handler is non-zero.
- `__attribute__((swift_async_error(nonnull_error)))`, specifies that the async method is considered to have failed if the `NSError *` argument to the completion handler is non-null.
- `__attribute__((swift_async_error(none)))`, specifies that the async method cannot fail.

For instance:

```
@interface MyClass : NSObject
-(void)asyncMethod:(void (^)(char, int, float))handler
    __attribute__((swift_async(swift_private, 1)))
    __attribute__((swift_async_error(zero_argument, 2)));
@end
```

Here, the `swift_async` attribute specifies that `handler` is the completion handler for this method, and the `swift_async_error` attribute specifies that the `int` parameter is the one that represents the error.

swift_async_name

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
swift_async_name							Yes

The `swift_async_name` attribute provides the name of the `async` overload for the given declaration in Swift. If this attribute is absent, the name is transformed according to the algorithm built into the Swift compiler.

The argument is a string literal that contains the Swift name of the function or method. The name may be a compound Swift name. The function or method with such an attribute must have more than zero parameters, as its last parameter is assumed to be a callback that's eliminated in the Swift `async` name.

```
@interface URL
+ (void) loadContentsFrom:(URL *)url callback:(void (^)(NSData *))data __attribute__((__swift_async_name__("URL.loadContentsFrom(_:)")))
@end
```

swift_attr

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
swift_attr							Yes

The `swift_attr` provides a Swift-specific annotation for the declaration to which the attribute appertains to. It can be used on any declaration in Clang. This kind of annotation is ignored by Clang as it doesn't have any semantic meaning in languages supported by Clang. The Swift compiler can interpret these annotations according to its own rules when importing C or Objective-C declarations.

swift_bridge

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
swift_bridge							

The `swift_bridge` attribute indicates that the declaration to which the attribute appertains is bridged to the named Swift type.

```
__attribute__((__objc_root__))
@interface Base
- (instancetype)init;
@end

__attribute__((__swift_bridge__("BridgedI")))
@interface I : Base
@end
```

In this example, the Objective-C interface `I` will be made available to Swift with the name `BridgedI`. It would be possible for the compiler to refer to `I` still in order to bridge the type back to Objective-C.

swift_bridged

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
swift_bridged_typedef							Yes

The `swift_bridged_typedef` attribute indicates that when the typedef to which the attribute appertains is imported into Swift, it should refer to the bridged Swift type (e.g. Swift's `String`) rather than the Objective-C type as written (e.g. `NSString`).

```
@interface NSString;
typedef NSString *AliasedString __attribute__((__swift_bridged_typedef__));

extern void acceptsAliasedString(AliasedString _Nonnull parameter);
```

In this case, the function `acceptsAliasedString` will be imported into Swift as a function which accepts a `String` type parameter.

swift_error

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
swift_error							Yes

The `swift_error` attribute controls whether a particular function (or Objective-C method) is imported into Swift as a throwing function, and if so, which dynamic convention it uses.

All of these conventions except `none` require the function to have an error parameter. Currently, the error parameter is always the last parameter of type `NSError**` or `CFErrorRef*`. Swift will remove the error parameter from the imported API. When calling the API, Swift will always pass a valid address initialized to a null pointer.

- `swift_error(none)` means that the function should not be imported as throwing. The error parameter and result type will be imported normally.
- `swift_error(null_result)` means that calls to the function should be considered to have thrown if they return a null value. The return type must be a pointer type, and it will be imported into Swift with a non-optional type. This is the default error convention for Objective-C methods that return pointers.
- `swift_error(zero_result)` means that calls to the function should be considered to have thrown if they return a zero result. The return type must be an integral type. If the return type would have been imported as `Bool`, it is instead imported as `Void`. This is the default error convention for Objective-C methods that return a type that would be imported as `Bool`.
- `swift_error(nonzero_result)` means that calls to the function should be considered to have thrown if they return a non-zero result. The return type must be an integral type. If the return type would have been imported as `Bool`, it is instead imported as `Void`.
- `swift_error(nonnull_error)` means that calls to the function should be considered to have thrown if they leave a non-null error in the error parameter. The return type is left unmodified.

swift_name

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
swift_name							

The `swift_name` attribute provides the name of the declaration in Swift. If this attribute is absent, the name is transformed according to the algorithm built into the Swift compiler.

The argument is a string literal that contains the Swift name of the function, variable, or type. When renaming a function, the name may be a compound Swift name. For a type, enum constant, property, or variable declaration, the name must be a simple or qualified identifier.

```
@interface URL
- (void) initWithString:(NSString *)s __attribute__((__swift_name__("URL.init(_:)")))
@end

void __attribute__((__swift_name__("squareRoot()"))) sqrt(double v) {
}
```

swift_newtype

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
swift_newtype swift_wrapper							Yes

The `swift_newtype` attribute indicates that the typedef to which the attribute appertains is imported as a new Swift type of the typedef's name. Previously, the attribute was spelt `swift_wrapper`. While the behaviour of the attribute is identical with either spelling, `swift_wrapper` is deprecated, only exists for compatibility purposes, and should not be used in new code.

- `swift_newtype(struct)` means that a Swift struct will be created for this typedef.
- `swift_newtype(enum)` means that a Swift enum will be created for this typedef.

```
// Import UIFontTextStyle as an enum type, with enumerated values being
// constants.
typedef NSString * UIFontTextStyle __attribute__((__swift_newtype__(enum)));

// Import UIFontDescriptorFeatureKey as a structure type, with enumerated
// values being members of the type structure.
typedef NSString * UIFontDescriptorFeatureKey __attribute__((__swift_newtype__(struct)));
```

swift_objc_members

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
swift_objc_members							Yes

This attribute indicates that Swift subclasses and members of Swift extensions of this class will be implicitly marked with the `@objcMembers` Swift attribute, exposing them back to Objective-C.

swift_private

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
swift_private							

Declarations marked with the `swift_private` attribute are hidden from the framework client but are still made available for use within the framework or Swift SDK overlay.

The purpose of this attribute is to permit a more idomatic implementation of declarations in Swift while hiding the non-idiomatic one.

Declaration Attributes

Owner

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
	gsl::Owner						Yes

Note

This attribute is experimental and its effect on analysis is subject to change in a future version of clang.

The attribute `[[gsl::Owner(T)]]` applies to structs and classes that own an object of type `T`:

```
class [[gsl::Owner(int)]] IntOwner {
private:
  int value;
public:
  int *getInt() { return &value; }
};
```

The argument `T` is optional and is ignored. This attribute may be used by analysis tools and has no effect on code generation. A `void` argument means that the class can own any type.

See Pointer for an example.

Pointer

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
	gsl::Pointer						Yes

Note

This attribute is experimental and its effect on analysis is subject to change in a future version of clang.

The attribute `[[gsl::Pointer(T)]]` applies to structs and classes that behave like pointers to an object of type `T`:

```
class [[gsl::Pointer(int)]] IntPointer {
private:
  int *valuePointer;
public:
  int *getInt() { return &valuePointer; }
};
```

The argument `T` is optional and is ignored. This attribute may be used by analysis tools and has no effect on code generation. A `void` argument means that the pointer can point to any type.

Example: When constructing an instance of a class annotated like this (a Pointer) from an instance of a class annotated with `[[gsl::Owner]]` (an Owner), then the analysis will consider the Pointer to point inside the Owner. When the Owner's lifetime ends, it will consider the Pointer to be dangling.

```
int f() {
  IntPointer P;
  if (true) {
    IntOwner O(7);
    P = IntPointer(O); // P "points into" O
  } // P is dangling
  return P.get(); // error: Using a dangling Pointer.
}
```

_Packed

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
				_Packed			

The z/OS compiler aligns structure and union members according to their natural byte boundaries and ends the structure or union on its natural boundary. However, since the alignment of a structure or union is that of the member with the largest alignment requirement, the compiler may add padding to elements whose byte boundaries are smaller than this requirement. You can use the _Packed qualifier to remove padding between members of structures or unions. Packed and nonpacked structures and unions have different storage layouts.

Consider the following example:

```
struct unpacked {
  int i;
  char c;
  short s;
};

_Packed union packed {
  int i;
  char c;
  short s;
};
```

In struct unpacked, since the largest alignment requirement among the members is that of int i, namely, 4 bytes, 3 bytes of padding are added at the end of char c (1 byte) and 2 bytes of padding are added at the end of short s (2 bytes).

In struct packed, there is no padding and each member takes exactly as many bytes as needed to represent the type.

__single_inhertiance, __multiple_inheritance, __virtual_inheritance

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
				__single_inheritance __multiple_inheritance __virtual_inheritance __unspecified_inheritance			

This collection of keywords is enabled under `-fms-extensions` and controls the pointer-to-member representation used on `*-*-win32` targets.

The `*-*-win32` targets utilize a pointer-to-member representation which varies in size and alignment depending on the definition of the underlying class.

However, this is problematic when a forward declaration is only available and no definition has been made yet. In such cases, Clang is forced to utilize the most general representation that is available to it.

These keywords make it possible to use a pointer-to-member representation other than the most general one regardless of whether or not the definition will ever be present in the current translation unit.

This family of keywords belong between the `class-key` and `class-name`:

```
struct __single_inheritance S;
int S::*i;
struct S {};
```

This keyword can be applied to class templates but only has an effect when used on full specializations:

```
template <typename T, typename U> struct __single_inheritance A; // warning: inheritance model ignored on primary template
template <typename T> struct __multiple_inheritance A<T, T>; // warning: inheritance model ignored on partial specialization
template <> struct __single_inheritance A<int, float>;
```

Note that choosing an inheritance model less general than strictly necessary is an error:

```
struct __multiple_inheritance S; // error: inheritance model does not match definition
int S::*i;
struct S {};
```

asm

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
				asm __asm__			

This attribute can be used on a function or variable to specify its symbol name.

On some targets, all C symbols are prefixed by default with a single character, typically _. This was done historically to distinguish them from symbols used by other languages. (This prefix is also added to the standard Itanium C++ ABI prefix on “mangled” symbol names, so that e.g. on such targets the true symbol name for a C++ variable declared as `int cppvar;` would be `__Z6cppvar`; note the two underscores.) This prefix is *not* added to the symbol names specified by the `asm` attribute; programmers wishing to match a C symbol name must compensate for this.

For example, consider the following C code:

```
int var1 asm("altvar") = 1;  // "altvar" in symbol table.
int var2 = 1; // "_var2" in symbol table.

void func1(void) asm("altfunc");
void func1(void) {} // "altfunc" in symbol table.
void func2(void) {} // "_func2" in symbol table.
```

Clang’s implementation of this attribute is compatible with GCC’s, documented here.

While it is possible to use this attribute to name a special symbol used internally by the compiler, such as an LLVM intrinsic, this is neither recommended nor supported and may cause the compiler to crash or miscompile. Users who wish to gain access to intrinsic behavior are strongly encouraged to request new builtin functions.

deprecated

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
deprecated	gnu::deprecated deprecated	gnu::deprecated deprecated	deprecated				

The `deprecated` attribute can be applied to a function, a variable, or a type. This is useful when identifying functions, variables, or types that are expected to be removed in a future version of a program.

Consider the function declaration for a hypothetical function `f`:

```
void f(void) __attribute__((deprecated("message", "replacement")));
```

When spelled as `__attribute__((deprecated))`, the deprecated attribute can have two optional string arguments. The first one is the message to display when emitting the warning; the second one enables the compiler to provide a Fix-It to replace the deprecated name with a new name. Otherwise, when spelled as `[[gnu::deprecated]]` or `[[deprecated]]`, the attribute can have one optional string argument which is the message to display when emitting the warning.

empty_bases

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
			empty_bases				

The empty_bases attribute permits the compiler to utilize the empty-base-optimization more frequently. This attribute only applies to struct, class, and union types. It is only supported when using the Microsoft C++ ABI.

enum_extensibility

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
enum_extensibility	clang::enum_extensibility	clang::enum_extensibility					Yes

Attribute `enum_extensibility` is used to distinguish between enum definitions that are extensible and those that are not. The attribute can take either `closed` or `open` as an argument. `closed` indicates a variable of the enum type takes a value that corresponds to one of the enumerators listed in the enum definition or, when the enum is annotated with `flag_enum`, a value that can be constructed using values corresponding to the enumerators. `open` indicates a variable of the enum type can take any values allowed by the standard and instructs clang to be more lenient when issuing warnings.

```
enum __attribute__((enum_extensibility(closed))) ClosedEnum {
  A0, A1
};

enum __attribute__((enum_extensibility(open))) OpenEnum {
  B0, B1
};

enum __attribute__((enum_extensibility(closed),flag_enum)) ClosedFlagEnum {
  C0 = 1 << 0, C1 = 1 << 1
};

enum __attribute__((enum_extensibility(open),flag_enum)) OpenFlagEnum {
  D0 = 1 << 0, D1 = 1 << 1
};

void foo1() {
  enum ClosedEnum ce;
  enum OpenEnum oe;
  enum ClosedFlagEnum cfe;
  enum OpenFlagEnum ofe;

  ce = A1;           // no warnings
  ce = 100;          // warning issued
  oe = B1;           // no warnings
  oe = 100;          // no warnings
  cfe = C0 | C1;     // no warnings
  cfe = C0 | C1 | 4; // warning issued
  ofe = D0 | D1;     // no warnings
  ofe = D0 | D1 | 4; // no warnings
}
```

external_source_symbol

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
external_source_symbol	clang::external_source_symbol	clang::external_source_symbol					Yes

The `external_source_symbol` attribute specifies that a declaration originates from an external source and describes the nature of that source.

The fact that Clang is capable of recognizing declarations that were defined externally can be used to provide better tooling support for mixed-language projects or projects that rely on auto-generated code. For instance, an IDE that uses Clang and that supports mixed-language projects can use this attribute to provide a correct 'jump-to-definition' feature. For a concrete example, consider a protocol that's defined in a Swift file:

```
@objc public protocol SwiftProtocol {
  func method()
}
```

This protocol can be used from Objective-C code by including a header file that was generated by the Swift compiler. The declarations in that header can use the `external_source_symbol` attribute to make Clang aware of the fact that `SwiftProtocol` actually originates from a Swift module:

```
__attribute__((external_source_symbol(language="Swift",defined_in="module")))
@protocol SwiftProtocol
@required
- (void) method;
@end
```

Consequently, when 'jump-to-definition' is performed at a location that references `SwiftProtocol`, the IDE can jump to the original definition in the Swift source file rather than jumping to the Objective-C declaration in the auto-generated header file.

The `external_source_symbol` attribute is a comma-separated list that includes clauses that describe the origin and the nature of the particular declaration. Those clauses can be:

language=*string-literal*

The name of the source language in which this declaration was defined.

defined_in=*string-literal*

The name of the source container in which the declaration was defined. The exact definition of source container is language-specific, e.g. Swift's source containers are modules, so `defined_in` should specify the Swift module name.

generated_declaration

This declaration was automatically generated by some tool.

The clauses can be specified in any order. The clauses that are listed above are all optional, but the attribute has to have at least one clause.

flag_enum

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
flag_enum	clang::flag_enum	clang::flag_enum					Yes

This attribute can be added to an enumerator to signal to the compiler that it is intended to be used as a flag type. This will cause the compiler to assume that the range of the type includes all of the values that you can get by manipulating bits of the enumerator when issuing warnings.

layout_version

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
			layout_version				

The layout_version attribute requests that the compiler utilize the class layout rules of a particular compiler version. This attribute only applies to struct, class, and union types. It is only supported when using the Microsoft C++ ABI.

lto_visibility_public

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
lto_visibility_public	clang::lto_visibility_public	clang::lto_visibility_public					Yes

See LTO Visibility.

managed

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
managed			__managed__				Yes

The __managed__ attribute can be applied to a global variable declaration in HIP. A managed variable is emitted as an undefined global symbol in the device binary and is registered by __hipRegisterManagedVariable in init functions. The HIP runtime allocates managed memory and uses it to define the symbol when loading the device binary. A managed variable can be accessed in both device and host code.

novtable

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
			novtable				

This attribute can be added to a class declaration or definition to signal to the compiler that constructors and destructors will not reference the virtual function table. It is only supported when using the Microsoft C++ ABI.

ns_error_domain

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
ns_error_domain							Yes

In Cocoa frameworks in Objective-C, one can group related error codes in enums and categorize these enums with error domains.

The `ns_error_domain` attribute indicates a global `NSString` or `CFString` constant representing the error domain that an error code belongs to. For pointer uniqueness and code size this is a constant symbol, not a literal.

The domain and error code need to be used together. The `ns_error_domain` attribute links error codes to their domain at the source level.

This metadata is useful for documentation purposes, for static analysis, and for improving interoperability between Objective-C and Swift. It is not used for code generation in Objective-C.

For example:

```
#define NS_ERROR_ENUM(_type, _name, _domain)  \
  enum _name : _type _name; enum __attribute__((ns_error_domain(_domain))) _name : _type

extern NSString *const MyErrorDomain;
typedef NS_ERROR_ENUM(unsigned char, MyErrorEnum, MyErrorDomain) {
  MyErrFirst,
  MyErrSecond,
};
```

objc_boxable

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
objc_boxable	clang::objc_boxable	clang::objc_boxable					Yes

Structs and unions marked with the `objc_boxable` attribute can be used with the Objective-C boxed expression syntax, `@(...)`.

Usage: `__attribute__((objc_boxable))`. This attribute can only be placed on a declaration of a trivially-copyable struct or union:

```
struct __attribute__((objc_boxable)) some_struct {
  int i;
};
union __attribute__((objc_boxable)) some_union {
  int i;
  float f;
};
typedef struct __attribute__((objc_boxable)) _some_struct some_struct;

// ...

some_struct ss;
NSValue *boxed = @(ss);
```

objc_direct

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
objc_direct	clang::objc_direct	clang::objc_direct					Yes

The `objc_direct` attribute can be used to mark an Objective-C method as being *direct*. A direct method is treated statically like an ordinary method, but dynamically it behaves more like a C function. This lowers some of the costs associated with the method but also sacrifices some of the ordinary capabilities of Objective-C methods.

A message send of a direct method calls the implementation directly, as if it were a C function, rather than using ordinary Objective-C method dispatch. This is substantially faster and potentially allows the implementation to be inlined, but it also means the method cannot be overridden in subclasses or replaced dynamically, as ordinary Objective-C methods can.

Furthermore, a direct method is not listed in the class's method lists. This substantially reduces the code-size overhead of the method but also means it cannot be called dynamically using ordinary Objective-C method dispatch at all; in particular, this means that it cannot override a superclass method or satisfy a protocol requirement.

Because a direct method cannot be overridden, it is an error to perform a `super` message send of one.

Although a message send of a direct method causes the method to be called directly as if it were a C function, it still obeys Objective-C semantics in other ways:

- If the receiver is `nil`, the message send does nothing and returns the zero value for the return type.
- A message send of a direct class method will cause the class to be initialized, including calling the `+initialize` method if present.
- The implicit `_cmd` parameter containing the method's selector is still defined. In order to minimize code-size costs, the implementation will not emit a reference to the selector if the parameter is unused within the method.

Symbols for direct method implementations are implicitly given hidden visibility, meaning that they can only be called within the same linkage unit.

It is an error to do any of the following:

- declare a direct method in a protocol,
- declare an override of a direct method with a method in a subclass,
- declare an override of a non-direct method with a direct method in a subclass,
- declare a method with different directness in different class interfaces, or
- implement a non-direct method (as declared in any class interface) with a direct method.

If any of these rules would be violated if every method defined in an `@implementation` within a single linkage unit were declared in an appropriate class interface, the program is ill-formed with no diagnostic required. If a violation of this rule is not diagnosed, behavior remains well-defined; this paragraph is simply reserving the right to diagnose such conflicts in the future, not to treat them as undefined behavior.

Additionally, Clang will warn about any `@selector` expression that names a selector that is only known to be used for direct methods.

For the purpose of these rules, a “class interface” includes a class’s primary `@interface` block, its class extensions, its categories, its declared protocols, and all the class interfaces of its superclasses.

An Objective-C property can be declared with the `direct` property attribute. If a direct property declaration causes an implicit declaration of a getter or setter method (that is, if the given method is not explicitly declared elsewhere), the method is declared to be direct.

Some programmers may wish to make many methods direct at once. In order to simplify this, the `objc_direct_members` attribute is provided; see its documentation for more information.

objc_direct_members

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
objc_direct_members	clang::objc_direct_members	clang::objc_direct_members					Yes

The `objc_direct_members` attribute can be placed on an Objective-C `@interface` or `@implementation` to mark that methods declared therein should be considered direct by default. See the documentation for `objc_direct` for more information about direct methods.

When `objc_direct_members` is placed on an `@interface` block, every method in the block is considered to be declared as direct. This includes any implicit method declarations introduced by property declarations. If the method redeclares a non-direct method, the declaration is ill-formed, exactly as if the method was annotated with the `objc_direct` attribute.

When `objc_direct_members` is placed on an `@implementation` block, methods defined in the block are considered to be declared as direct unless they have been previously declared as non-direct in any interface of the class. This includes the implicit method definitions introduced by synthesized properties, including auto-synthesized properties.

objc_non_runtime_protocol

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
objc_non_runtime_protocol	clang::objc_non_runtime_protocol	clang::objc_non_runtime_protocol					Yes

The `objc_non_runtime_protocol` attribute can be used to mark that an Objective-C protocol is only used during static type-checking and doesn’t need to be represented dynamically. This avoids several small code-size and run-time overheads associated with handling the protocol’s metadata. A non-runtime protocol cannot be used as the operand of a `@protocol` expression, and dynamic attempts to find it with `objc_getProtocol` will fail.

If a non-runtime protocol inherits from any ordinary protocols, classes and derived protocols that declare conformance to the non-runtime protocol will dynamically list their conformance to those bare protocols.

objc_nonlazy_class

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
objc_nonlazy_class	clang::objc_nonlazy_class	clang::objc_nonlazy_class					Yes

This attribute can be added to an Objective-C `@interface` or `@implementation` declaration to add the class to the list of non-lazily initialized classes. A non-lazy class will be initialized eagerly when the Objective-C runtime is loaded. This is required for certain system classes which have instances allocated in non-standard ways, such as the classes for blocks and constant strings. Adding this attribute is essentially equivalent to providing a trivial `+load` method but avoids the (fairly small) load-time overheads associated with defining and calling such a method.

objc_runtime_name

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
objc_runtime_name	clang::objc_runtime_name	clang::objc_runtime_name					Yes

By default, the Objective-C interface or protocol identifier is used in the metadata name for that object. The `objc_runtime_name` attribute allows annotated interfaces or protocols to use the specified string argument in the object's metadata name instead of the default name.

Usage: `__attribute__((objc_runtime_name("MyLocalName")))`. This attribute can only be placed before an @protocol or @interface declaration:

```
__attribute__((objc_runtime_name("MyLocalName")))
@interface Message
@end
```

objc_runtime_visible

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
objc_runtime_visible	clang::objc_runtime_visible	clang::objc_runtime_visible					Yes

This attribute specifies that the Objective-C class to which it applies is visible to the Objective-C runtime but not to the linker. Classes annotated with this attribute cannot be subclassed and cannot have categories defined for them.

objc_subclassing_restricted

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
objc_subclassing_restricted	clang::objc_subclassing_restricted	clang::objc_subclassing_restricted					Yes

This attribute can be added to an Objective-C `@interface` declaration to ensure that this class cannot be subclassed.

preferred_name

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
preferred_name	clang::preferred_name						

The `preferred_name` attribute can be applied to a class template, and specifies a preferred way of naming a specialization of the template. The preferred name will be used whenever the corresponding template specialization would otherwise be printed in a diagnostic or similar context.

The preferred name must be a typedef or type alias declaration that refers to a specialization of the class template (not including any type qualifiers). In general this requires the template to be declared at least twice. For example:

```
template<typename T> struct basic_string;
using string = basic_string<char>;
using wstring = basic_string<wchar_t>;
template<typename T> struct [[clang::preferred_name(string),
                              clang::preferred_name(wstring)]] basic_string {
  // ...
};
```

randomize_layout, no_randomize_layout

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
no_randomize_layout	gnu::no_randomize_layout	gnu::no_randomize_layout					Yes

The attribute `randomize_layout`, when attached to a C structure, selects it for structure layout field randomization; a compile-time hardening technique. A “seed” value, is specified via the `-frandomize-layout-seed=` command line flag. For example:

```
SEED=`od -A n -t x8 -N 32 /dev/urandom | tr -d ' \n'`
make ... CFLAGS="-frandomize-layout-seed=$SEED" ...
```

You can also supply the seed in a file with `-frandomize-layout-seed-file=`. For example:

```
od -A n -t x8 -N 32 /dev/urandom | tr -d ' \n' > /tmp/seed_file.txt
make ... CFLAGS="-frandomize-layout-seed-file=/tmp/seed_file.txt" ...
```

The randomization is deterministic based for a given seed, so the entire program should be compiled with the same seed, but keep the seed safe otherwise.

The attribute `no_randomize_layout`, when attached to a C structure, instructs the compiler that this structure should not have its field layout randomized.

randomize_layout, no_randomize_layout

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
randomize_layout	gnu::randomize_layout	gnu::randomize_layout					Yes

The attribute `randomize_layout`, when attached to a C structure, selects it for structure layout field randomization; a compile-time hardening technique. A “seed” value, is specified via the `-frandomize-layout-seed=` command line flag. For example:

```
SEED=`od -A n -t x8 -N 32 /dev/urandom | tr -d ' \n'`
make ... CFLAGS="-frandomize-layout-seed=$SEED" ...
```

You can also supply the seed in a file with `-frandomize-layout-seed-file=`. For example:

```
od -A n -t x8 -N 32 /dev/urandom | tr -d ' \n' > /tmp/seed_file.txt
make ... CFLAGS="-frandomize-layout-seed-file=/tmp/seed_file.txt" ...
```

The randomization is deterministic based for a given seed, so the entire program should be compiled with the same seed, but keep the seed safe otherwise.

The attribute `no_randomize_layout`, when attached to a C structure, instructs the compiler that this structure should not have its field layout randomized.

selectany

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
selectany	gnu::selectany	gnu::selectany	selectany				

This attribute appertains to a global symbol, causing it to have a weak definition (linkonce), allowing the linker to select any definition.

For more information see gcc documentation or msvc documentation.

transparent_union

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
transparent_union	gnu::transparent_union	gnu::transparent_union					

This attribute can be applied to a union to change the behavior of calls to functions that have an argument with a transparent union type. The compiler behavior is changed in the following manner:

- A value whose type is any member of the transparent union can be passed as an argument without the need to cast that value.
- The argument is passed to the function using the calling convention of the first member of the transparent union. Consequently, all the members of the transparent union should have the same calling convention as its first member.

Transparent unions are not supported in C++.

trivial_abi

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
trivial_abi	clang::trivial_abi						Yes

The `trivial_abi` attribute can be applied to a C++ class, struct, or union. It instructs the compiler to pass and return the type using the C ABI for the underlying type when the type would otherwise be considered non-trivial for the purpose of calls. A class annotated with `trivial_abi` can have non-trivial destructors or copy/move constructors without automatically becoming non-trivial for the purposes of calls. For example:

```
// A is trivial for the purposes of calls because ``trivial_abi`` makes the
// user-provided special functions trivial.
struct __attribute__((trivial_abi)) A {
  ~A();
  A(const A &);
  A(A &&);
  int x;
};

// B's destructor and copy/move constructor are considered trivial for the
// purpose of calls because A is trivial.
struct B {
  A a;
};
```

If a type is trivial for the purposes of calls, has a non-trivial destructor, and is passed as an argument by value, the convention is that the callee will destroy the object before returning.

If a type is trivial for the purpose of calls, it is assumed to be trivially relocatable for the purpose of `__is_trivially_relocatable`.

Attribute `trivial_abi` has no effect in the following cases:

- The class directly declares a virtual base or virtual methods.
- Copy constructors and move constructors of the class are all deleted.

- The class has a base class that is non-trivial for the purposes of calls.
- The class has a non-static data member whose type is non-trivial for the purposes of calls, which includes:
  - classes that are non-trivial for the purposes of calls
  - __weak-qualified types in Objective-C++
  - arrays of any of the above

using_if_exists

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
using_if_exists	clang::using_if_exists						

The `using_if_exists` attribute applies to a using-declaration. It allows programmers to import a declaration that potentially does not exist, instead deferring any errors to the point of use. For instance:

```
namespace empty_namespace {};
__attribute__((using_if_exists))
using empty_namespace::does_not_exist; // no error!

does_not_exist x; // error: use of unresolved 'using_if_exists'
```

The C++ spelling of the attribte (*[[clang::using_if_exists]]*) is also supported as a clang extension, since ISO C++ doesn't support attributes in this position. If the entity referred to by the using-declaration is found by name lookup, the attribute has no effect. This attribute is useful for libraries (primarily, libc++) that wish to redeclare a set of declarations in another namespace, when the availability of those declarations is difficult or impossible to detect at compile time with the preprocessor.

Field Attributes

no_unique_address

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
	no_unique_address						

The `no_unique_address` attribute allows tail padding in a non-static data member to overlap other members of the enclosing class (and in the special case when the type is empty, permits it to fully overlap other members). The field is laid out as if a base class were encountered at the corresponding point within the class (except that it does not share a vptr with the enclosing object).

Example usage:

```
template<typename T, typename Alloc> struct my_vector {
  T *p;
  [[no_unique_address]] Alloc alloc;
  // ...
```

```
};
static_assert(sizeof(my_vector<int, std::allocator<int>>) == sizeof(int*));
```

`[[no_unique_address]]` is a standard C++20 attribute. Clang supports its use in C++11 onwards.

Function Attributes

#pragma omp declare simd

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
					omp declare simd		

The `declare simd` construct can be applied to a function to enable the creation of one or more versions that can process multiple arguments using SIMD instructions from a single invocation in a SIMD loop. The `declare simd` directive is a declarative directive. There may be multiple `declare simd` directives for a function. The use of a `declare simd` construct on a function enables the creation of SIMD versions of the associated function that can be used to process multiple arguments from a single invocation from a SIMD loop concurrently. The syntax of the `declare simd` construct is as follows:

```
#pragma omp declare simd [clause[[,] clause] ...] new-line
[#pragma omp declare simd [clause[[,] clause] ...] new-line]
[...]
function definition or declaration
```

where clause is one of the following:

```
simdlen(length)
linear(argument-list[:constant-linear-step])
aligned(argument-list[:alignment])
uniform(argument-list)
inbranch
notinbranch
```

#pragma omp declare target

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
					omp declare target		

The `declare target` directive specifies that variables and functions are mapped to a device for OpenMP offload mechanism.

The syntax of the declare target directive is as follows:

```
#pragma omp declare target new-line
declarations-definition-seq
#pragma omp end declare target new-line
```

or

```
#pragma omp declare target (extended-list) new-line
```

or

```
#pragma omp declare target clause[ [,] clause ... ] new-line
```

where clause is one of the following:

```
to(extended-list)
link(list)
device_type(host | nohost | any)
```

#pragma omp declare variant

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
					omp declare variant		

The `declare variant` directive declares a specialized variant of a base function and specifies the context in which that specialized variant is used. The declare variant directive is a declarative directive. The syntax of the `declare variant` construct is as follows:

```
#pragma omp declare variant(variant-func-id) clause new-line
[#pragma omp declare variant(variant-func-id) clause new-line]
[...]
function definition or declaration
```

where clause is one of the following:

```
match(context-selector-specification)
```

and where `variant-func-id` is the name of a function variant that is either a base language identifier or, for C++, a template-id.

Clang provides the following context selector extensions, used via `implementation={extension(EXTENSION)}`:

```
match_all
match_any
match_none
disable_implicit_base
allow_templates
```

The match extensions change when the *entire* context selector is considered a match for an OpenMP context. The default is `all`, with `none` no trait in the selector is allowed to be in the OpenMP context, with `any` a single trait in both the selector and OpenMP context is sufficient. Only a single match extension trait is allowed per context selector. The disable extensions remove default effects of the `begin declare variant` applied to a definition. If `disable_implicit_base` is given, we will not introduce an implicit base function for a variant if no base function was found. The variant is still generated but will never be called, due to the absence of a base function and consequently calls to a base function. The allow extensions change when the `begin declare variant` effect is applied to a definition. If `allow_templates` is given, template function definitions are considered as specializations of existing or assumed template declarations with the same name. The template parameters for the base functions are used to instantiate the specialization.

SV_GroupIndex

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
						SV_GroupIndex	

The SV_GroupIndex semantic, when applied to an input parameter, specifies a data binding to map the group index to the specified parameter. This attribute is only supported in compute shaders.

The full documentation is available here: https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/sv-groupindex

_Export

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
				_Export			

Use the _Export keyword with a function name or external variable to declare that it is to be exported (made available to other modules). You must define the object name in the same translation unit in which you use the _Export keyword. For example:

```
int _Export anthony(float);
```

This statement exports the function anthony, if you define the function in the translation unit. The _Export keyword must immediately precede the object name. If you apply the _Export keyword to a class, the compiler automatically exports all static data members and member functions of the class. However, if you want it to apply to individual class members, then you must apply it to each member that can be referenced.

_Noreturn

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
				_Noreturn			

A function declared as _Noreturn shall not return to its caller. The compiler will generate a diagnostic for a function declared as _Noreturn that appears to be capable of returning to its caller. Despite being a type specifier, the _Noreturn attribute cannot be specified on a function pointer type.

abi_tag

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
abi_tag	gnu::abi_tag						Yes

The `abi_tag` attribute can be applied to a function, variable, class or inline namespace declaration to modify the mangled name of the entity. It gives the ability to distinguish between different versions of the same entity but with different ABI versions supported. For example, a newer version of a class could have a different set of data members and thus have a different size. Using the `abi_tag` attribute, it is possible to have different mangled names for a global variable of the class type. Therefore, the old code could keep using the old mangled name and the new code will use the new mangled name with tags.

acquire_capability, acquire_shared_capability

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
acquire_capability acquire_shared_capability exclusive_lock_function shared_lock_function	clang::acquire_capability clang::acquire_shared_capability						

Marks a function as acquiring a capability.

alloc_align

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
alloc_align	gnu::alloc_align	gnu::alloc_align					

Use `__attribute__((alloc_align(<alignment>))` on a function declaration to specify that the return value of the function (which must be a pointer type) is at least as aligned as the value of the indicated parameter. The parameter is given by its index in the list of formal parameters; the first parameter has index 1 unless the function is a C++ non-static member function, in which case the first parameter has index 2 to account for the implicit `this` parameter.

```
// The returned pointer has the alignment specified by the first parameter.
void *a(size_t align) __attribute__((alloc_align(1)));

// The returned pointer has the alignment specified by the second parameter.
void *b(void *v, size_t align) __attribute__((alloc_align(2)));
```

```
// The returned pointer has the alignment specified by the second visible
// parameter, however it must be adjusted for the implicit 'this' parameter.
void *Foo::b(void *v, size_t align) __attribute__((alloc_align(3)));
```

Note that this attribute merely informs the compiler that a function always returns a sufficiently aligned pointer. It does not cause the compiler to emit code to enforce that alignment. The behavior is undefined if the returned pointer is not sufficiently aligned.

alloc_size

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
alloc_size	gnu::alloc_size	gnu::alloc_size					

The `alloc_size` attribute can be placed on functions that return pointers in order to hint to the compiler how many bytes of memory will be available at the returned pointer. `alloc_size` takes one or two arguments.

- `alloc_size(N)` implies that argument number N equals the number of available bytes at the returned pointer.
- `alloc_size(N, M)` implies that the product of argument number N and argument number M equals the number of available bytes at the returned pointer.

Argument numbers are 1-based.

An example of how to use `alloc_size`

```
void *my_malloc(int a) __attribute__((alloc_size(1)));
void *my_calloc(int a, int b) __attribute__((alloc_size(1, 2)));

int main() {
  void *const p = my_malloc(100);
  assert(__builtin_object_size(p, 0) == 100);
  void *const a = my_calloc(20, 5);
  assert(__builtin_object_size(a, 0) == 100);
}
```

Note

This attribute works differently in clang than it does in GCC. Specifically, clang will only trace `const` pointers (as above); we give up on pointers that are not marked as `const`. In the vast majority of cases, this is unimportant, because LLVM has support for the `alloc_size` attribute. However, this may cause mildly unintuitive behavior when used with other attributes, such as `enable_if`.

allocator

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
			allocator				

The `__declspec(allocator)` attribute is applied to functions that allocate memory, such as operator new in C++. When CodeView debug information is emitted (enabled by `clang -gcodeview` or `clang-cl /Z7`), Clang will attempt to record the code offset of heap allocation call sites in the debug info. It will also record the type being allocated using some local heuristics. The Visual Studio debugger uses this information to profile memory usage.

This attribute does not affect optimizations in any way, unlike GCC's `__attribute__((malloc))`.

always_inline, __force_inline

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
always_inline	gnu::always_inline clang::always_inline	gnu::always_inline clang::always_inline		__forceinline			Yes

Inlining heuristics are disabled and inlining is always attempted regardless of optimization level.

`[[clang::always_inline]]` spelling can be used as a statement attribute; other spellings of the attribute are not supported on statements. If a statement is marked `[[clang::always_inline]]` and contains calls, the compiler attempts to inline those calls.

```
int example(void) {
  int i;
  [[clang::always_inline]] foo(); // attempts to inline foo
  [[clang::always_inline]] i = bar(); // attempts to inline bar
  [[clang::always_inline]] return f(42, baz(bar())); // attempts to inline everything
}
```

A declaration statement, which is a statement, is not a statement that can have an attribute associated with it (the attribute applies to the declaration, not the statement in that case). So this use case will not work:

```
int example(void) {
  [[clang::always_inline]] int i = bar();
  return i;
}
```

This attribute does not guarantee that inline substitution actually occurs.

<ins>Note: applying this attribute to a coroutine at the *-O0* optimization level has no effect; other optimization levels may only partially inline and result in a diagnostic.</ins>

See also the Microsoft Docs on Inline Functions, the GCC Common Function Attribute docs, and the GCC Inline docs.

artificial

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
artificial	gnu::artificial	gnu::artificial					

The `artificial` attribute can be applied to an inline function. If such a function is inlined, the attribute indicates that debuggers should associate the resulting instructions with the call site, rather than with the corresponding line within the inlined callee.

assert_capability, assert_shared_capability

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
assert_capability assert_shared_capability	clang::assert_capability clang::assert_shared_capability						

Marks a function that dynamically tests whether a capability is held, and halts the program if it is not held.

assume

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
assume	clang::assume	clang::assume					Yes

Clang supports the `__attribute__((assume("assumption")))` attribute to provide additional information to the optimizer. The string-literal, here “assumption”, will be attached to the function declaration such that later analysis and optimization passes can assume the “assumption” to hold. This is similar to __builtin_assume but instead of an expression that can be assumed to be non-zero, the assumption is expressed as a string and it holds for the entire function.

A function can have multiple assume attributes and they propagate from prior declarations to later definitions. Multiple assumptions are aggregated into a single comma separated string. Thus, one can provide multiple assumptions via a comma separated string, i.a., `__attribute__((assume("assumption1,assumption2")))`.

While LLVM plugins might provide more assumption strings, the default LLVM optimization passes are aware of the following assumptions:

```
"omp_no_openmp"
"omp_no_openmp_routines"
"omp_no_parallelism"
```

The OpenMP standard defines the meaning of OpenMP assumptions (“omp_XYZ” is spelled “XYZ” in the OpenMP 5.1 Standard).

assume_aligned

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
assume_aligned	gnu::assume_aligned	gnu::assume_aligned					Yes

Use `__attribute__((assume_aligned(<alignment>[,<offset>]))` on a function declaration to specify that the return value of the function (which must be a pointer type) has the specified offset, in bytes, from an address with the specified alignment. The offset is taken to be zero if omitted.

```
// The returned pointer value has 32-byte alignment.
void *a() __attribute__((assume_aligned (32)));

// The returned pointer value is 4 bytes greater than an address having
// 32-byte alignment.
void *b() __attribute__((assume_aligned (32, 4)));
```

Note that this attribute provides information to the compiler regarding a condition that the code already ensures is true. It does not cause the compiler to enforce the provided alignment assumption.

availability

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
availability	clang::availability	clang::availability					Yes

The `availability` attribute can be placed on declarations to describe the lifecycle of that declaration relative to operating system versions. Consider the function declaration for a hypothetical function `f`:

```
void f(void) __attribute__((availability(macos,introduced=10.4,deprecated=10.6,obsoleted=10.7)));
```

The availability attribute states that `f` was introduced in macOS 10.4, deprecated in macOS 10.6, and obsoleted in macOS 10.7. This information is used by Clang to determine when it is safe to use `f`: for example, if Clang is instructed to compile code for macOS 10.5, a call to `f()` succeeds. If Clang is instructed to compile code for macOS 10.6, the call succeeds but Clang emits a warning specifying that the function is deprecated. Finally, if Clang is instructed to compile code for macOS 10.7, the call fails because `f()` is no longer available.

The availability attribute is a comma-separated list starting with the platform name and then including clauses specifying important milestones in the declaration's lifetime (in any order) along with additional information. Those clauses can be:

introduced=*version*

The first version in which this declaration was introduced.

deprecated=*version*

The first version in which this declaration was deprecated, meaning that users should migrate away from this API.

obsoleted=*version*

The first version in which this declaration was obsoleted, meaning that it was removed completely and can no longer be used.

unavailable

This declaration is never available on this platform.

message=*string-literal*

Additional message text that Clang will provide when emitting a warning or error about use of a deprecated or obsoleted declaration. Useful to direct users to replacement APIs.

replacement=*string-literal*

Additional message text that Clang will use to provide Fix-It when emitting a warning about use of a deprecated declaration. The Fix-It will replace the deprecated declaration with the new declaration specified.

Multiple availability attributes can be placed on a declaration, which may correspond to different platforms. For most platforms, the availability attribute with the platform corresponding to the target platform will be used; any others will be ignored. However, the availability for `watchOS` and `tvOS` can be implicitly inferred from an `iOS` availability attribute. Any explicit availability attributes for those platforms are still preferred over the implicitly inferred availability attributes. If no availability attribute specifies availability for the current target platform, the availability attributes are ignored. Supported platforms are:

`ios`

Apple's iOS operating system. The minimum deployment target is specified by the `-mios-version-min=*version*` or `-miphoneos-version-min=*version*` command-line arguments.

`macos`

Apple's macOS operating system. The minimum deployment target is specified by the `-mmacosx-version-min=*version*` command-line argument. `macosx` is supported for backward-compatibility reasons, but it is deprecated.

`tvos`

Apple's tvOS operating system. The minimum deployment target is specified by the `-mtvos-version-min=*version*` command-line argument.

`watchos`

Apple's watchOS operating system. The minimum deployment target is specified by the `-mwatchos-version-min=*version*` command-line argument.

`driverkit`

Apple's DriverKit userspace kernel extensions. The minimum deployment target is specified as part of the triple.

A declaration can typically be used even when deploying back to a platform version prior to when the declaration was introduced. When this happens, the declaration is weakly linked, as if the `weak_import` attribute were added to the declaration. A weakly-linked declaration may or may not be present a run-time, and a program can determine whether the declaration is present by checking whether the address of that declaration is non-NULL.

The flag `strict` disallows using API when deploying back to a platform version prior to when the declaration was introduced. An attempt to use such API before its introduction causes a hard error. Weakly-linking is almost always a better API choice, since it allows users to query availability at runtime.

If there are multiple declarations of the same entity, the availability attributes must either match on a per-platform basis or later declarations must not have availability attributes for that platform. For example:

```
void g(void) __attribute__((availability(macos,introduced=10.4)));
void g(void) __attribute__((availability(macos,introduced=10.4))); // okay, matches
void g(void) __attribute__((availability(ios,introduced=4.0))); // okay, adds a new platform
void g(void); // okay, inherits both macos and ios availability from above.
void g(void) __attribute__((availability(macos,introduced=10.5))); // error: mismatch
```

When one method overrides another, the overriding method can be more widely available than the overridden method, e.g.,:

```
@interface A
- (id)method __attribute__((availability(macos,introduced=10.4)));
- (id)method2 __attribute__((availability(macos,introduced=10.4)));
@end

@interface B : A
- (id)method __attribute__((availability(macos,introduced=10.3))); // okay: method moved into base class later
- (id)method __attribute__((availability(macos,introduced=10.5))); // error: this method was available via the base class in 10.4
@end
```

Starting with the macOS 10.12 SDK, the `API_AVAILABLE` macro from `<os/availability.h>` can simplify the spelling:

```
@interface A
- (id)method API_AVAILABLE(macos(10.11));
```

```
- (id)otherMethod API_AVAILABLE(macos(10.11), ios(11.0));
@end
```

Availability attributes can also be applied using a `#pragma clang attribute`. Any explicit availability attribute whose platform corresponds to the target platform is applied to a declaration regardless of the availability attributes specified in the pragma. For example, in the code below, `hasExplicitAvailabilityAttribute` will use the `macOS` availability attribute that is specified with the declaration, whereas `getsThePragmaAvailabilityAttribute` will use the `macOS` availability attribute that is applied by the pragma.

```
#pragma clang attribute push (__attribute__((availability(macOS, introduced=10.12))), apply_to=function)
void getsThePragmaAvailabilityAttribute(void);
void hasExplicitAvailabilityAttribute(void) __attribute__((availability(macos,introduced=10.4)));
#pragma clang attribute pop
```

For platforms like `watchOS` and `tvOS`, whose availability attributes can be implicitly inferred from an `iOS` availability attribute, the logic is slightly more complex. The explicit and the pragma-applied availability attributes whose platform corresponds to the target platform are applied as described in the previous paragraph. However, the implicitly inferred attributes are applied to a declaration only when there is no explicit or pragma-applied availability attribute whose platform corresponds to the target platform. For example, the function below will receive the `tvOS` availability from the pragma rather than using the inferred `iOS` availability from the declaration:

```
#pragma clang attribute push (__attribute__((availability(tvOS, introduced=12.0))), apply_to=function)
void getsThePragmaTVOSAvailabilityAttribute(void) __attribute__((availability(iOS,introduced=11.0)));
#pragma clang attribute pop
```

The compiler is also able to apply implicitly inferred attributes from a pragma as well. For example, when targeting `tvOS`, the function below will receive a `tvOS` availability attribute that is implicitly inferred from the `iOS` availability attribute applied by the pragma:

```
#pragma clang attribute push (__attribute__((availability(iOS, introduced=12.0))), apply_to=function)
void infersTVOSAvailabilityFromPragma(void);
#pragma clang attribute pop
```

The implicit attributes that are inferred from explicitly specified attributes whose platform corresponds to the target platform are applied to the declaration even if there is an availability attribute that can be inferred from a pragma. For example, the function below will receive the `tvOS, introduced=11.0` availability that is inferred from the attribute on the declaration rather than inferring availability from the pragma:

```
#pragma clang attribute push (__attribute__((availability(iOS, unavailable))), apply_to=function)
void infersTVOSAvailabilityFromAttributeNextToDeclaration(void)
  __attribute__((availability(iOS,introduced=11.0)));
#pragma clang attribute pop
```

Also see the documentation for @available

btf_decl_tag

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
btf_decl_tag	clang::btf_decl_tag	clang::btf_decl_tag					Yes

Clang supports the `__attribute__((btf_decl_tag("ARGUMENT")))` attribute for all targets. This attribute may be attached to a struct/union, struct/union field, function, function parameter, variable or typedef declaration. If `-g` is specified, the `ARGUMENT` info will be preserved in IR and be emitted to dwarf. For BPF targets, the `ARGUMENT` info will be emitted to .BTF ELF section too.

callback

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
callback	clang::callback	clang::callback					Yes

The `callback` attribute specifies that the annotated function may invoke the specified callback zero or more times. The callback, as well as the passed arguments, are identified by their parameter name or position (starting with 1!) in the annotated function. The first position in the attribute identifies the callback callee, the following positions declare describe its arguments. The callback callee is required to be callable with the number, and order, of the specified arguments. The index `0`, or the identifier `this`, is used to represent an implicit "this" pointer in class methods. If there is no implicit "this" pointer it shall not be referenced. The index '-1', or the name "__", represents an unknown callback callee argument. This can be a value which is not present in the declared parameter list, or one that is, but is potentially inspected, captured, or modified. Parameter names and indices can be mixed in the callback attribute.

The `callback` attribute, which is directly translated to `callback` metadata <http://llvm.org/docs/LangRef.html#callback-metadata>, make the connection between the call to the annotated function and the callback callee. This can enable interprocedural optimizations which were otherwise impossible. If a function parameter is mentioned in the `callback` attribute, through its position, it is undefined if that parameter is used for anything other than the actual callback. Inspected, captured, or modified parameters shall not be listed in the `callback` metadata.

Example encodings for the callback performed by `pthread_create` are shown below. The explicit attribute annotation indicates that the third parameter (`start_routine`) is called zero or more times by the `pthread_create` function, and that the fourth parameter (`arg`) is passed along. Note that the callback behavior of `pthread_create` is automatically recognized by Clang. In addition, the declarations of `__kmpc_fork_teams` and `__kmpc_fork_call`, generated for `#pragma omp target teams` and `#pragma omp parallel`, respectively, are also automatically recognized as broker functions. Further functions might be added in the future.

```
__attribute__((callback (start_routine, arg)))
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);

__attribute__((callback (3, 4)))
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

carries_dependency

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
carries_dependency	carries_dependency						Yes

The `carries_dependency` attribute specifies dependency propagation into and out of functions.

When specified on a function or Objective-C method, the `carries_dependency` attribute means that the return value carries a dependency out of the function, so that the implementation need not constrain ordering upon return from that function. Implementations of the function and its caller may choose to preserve dependencies instead of emitting memory ordering instructions such as fences.

Note, this attribute does not change the meaning of the program, but may result in generation of more efficient code.

cf_consumed

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
cf_consumed	clang::cf_consumed	clang::cf_consumed					Yes

The behavior of a function with respect to reference counting for Foundation (Objective-C), CoreFoundation (C) and OSObject (C++) is determined by a naming convention (e.g. functions starting with “get” are assumed to return at +0).

It can be overridden using a family of the following attributes. In Objective-C, the annotation `__attribute__((ns_returns_retained))` applied to a function communicates that the object is returned at +1, and the caller is responsible for freeing it. Similarly, the annotation `__attribute__((ns_returns_not_retained))` specifies that the object is returned at +0 and the ownership remains with the callee. The annotation `__attribute__((ns_consumes_self))` specifies that the Objective-C method call consumes the reference to `self`, e.g. by attaching it to a supplied parameter. Additionally, parameters can have an annotation `__attribute__((ns_consumed))`, which specifies that passing an owned object as that parameter effectively transfers the ownership, and the caller is no longer responsible for it. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

In C programs using CoreFoundation, a similar set of attributes: `__attribute__((cf_returns_not_retained))`, `__attribute__((cf_returns_retained))` and `__attribute__((cf_consumed))` have the same respective semantics when applied to CoreFoundation objects. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

Finally, in C++ interacting with XNU kernel (objects inheriting from OSObject), the same attribute family is present: `__attribute__((os_returns_not_retained))`, `__attribute__((os_returns_retained))` and `__attribute__((os_consumed))`, with the same respective semantics. Similar to `__attribute__((ns_consumes_self))`, `__attribute__((os_consumes_this))` specifies that the method call consumes the reference to “this” (e.g., when attaching it to a different object supplied as a parameter). Out parameters (parameters the function is meant to write into, either via pointers-to-pointers or references-to-pointers) may be annotated with `__attribute__((os_returns_retained))` or `__attribute__((os_returns_not_retained))` which specifies that the object written into the out parameter should (or respectively should not) be released after use. Since often out parameters may or may not be written depending on the exit code of the function, annotations `__attribute__((os_returns_retained_on_zero))` and `__attribute__((os_returns_retained_on_non_zero))` specify that an out parameter at +1 is written if and only if the function returns a zero (respectively non-zero) error code. Observe that return-code-dependent out parameter annotations are only available for retained out parameters, as non-retained object do not have to be released by the callee. These attributes are only used by the Clang Static Analyzer.

The family of attributes `X_returns_X_retained` can be added to functions, C++ methods, and Objective-C methods and properties. Attributes `X_consumed` can be added to parameters of methods, functions, and Objective-C methods.

cf_returns_not_retained

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
cf_returns_not_retained	clang::cf_returns_not_retained	clang::cf_returns_not_retained					

The behavior of a function with respect to reference counting for Foundation (Objective-C), CoreFoundation (C) and OSObject (C++) is determined by a naming convention (e.g. functions starting with “get” are assumed to return at `+0`).

It can be overridden using a family of the following attributes. In Objective-C, the annotation `__attribute__((ns_returns_retained))` applied to a function communicates that the object is returned at `+1`, and the caller is responsible for freeing it. Similarly, the annotation `__attribute__((ns_returns_not_retained))` specifies that the object is returned at `+0` and the ownership remains with the callee. The annotation `__attribute__((ns_consumes_self))` specifies that the Objective-C method call consumes the reference to `self`, e.g. by attaching it to a supplied parameter. Additionally, parameters can have an annotation `__attribute__((ns_consumed))`, which specifies that passing an owned object as that parameter effectively transfers the ownership, and the caller is no longer responsible for it. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

In C programs using CoreFoundation, a similar set of attributes: `__attribute__((cf_returns_not_retained))`, `__attribute__((cf_returns_retained))` and `__attribute__((cf_consumed))` have the same respective semantics when applied to CoreFoundation objects. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

Finally, in C++ interacting with XNU kernel (objects inheriting from OSObject), the same attribute family is present: `__attribute__((os_returns_not_retained))`, `__attribute__((os_returns_retained))` and `__attribute__((os_consumed))`, with the same respective semantics. Similar to `__attribute__((ns_consumes_self))`, `__attribute__((os_consumes_this))` specifies that the method call consumes the reference to “this” (e.g., when attaching it to a different object supplied as a parameter). Out parameters (parameters the function is meant to write into, either via pointers-to-pointers or references-to-pointers) may be annotated with `__attribute__((os_returns_retained))` or `__attribute__((os_returns_not_retained))` which specifies that the object written into the out parameter should (or respectively should not) be released after use. Since often out parameters may or may not be written depending on the exit code of the function, annotations `__attribute__((os_returns_retained_on_zero))` and `__attribute__((os_returns_retained_on_non_zero))` specify that an out parameter at `+1` is written if and only if the function returns a zero (respectively non-zero) error code. Observe that return-code-dependent out parameter annotations are only available for retained out parameters, as non-retained object do not have to be released by the callee. These attributes are only used by the Clang Static Analyzer.

The family of attributes `X_returns_X_retained` can be added to functions, C++ methods, and Objective-C methods and properties. Attributes `X_consumed` can be added to parameters of methods, functions, and Objective-C methods.

cf_returns_retained

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
cf_returns_retained	clang::cf_returns_retained	clang::cf_returns_retained					

The behavior of a function with respect to reference counting for Foundation (Objective-C), CoreFoundation (C) and OSObject (C++) is determined by a naming convention (e.g. functions starting with “get” are assumed to return at `+0`).

It can be overridden using a family of the following attributes. In Objective-C, the annotation `__attribute__((ns_returns_retained))` applied to a function communicates that the object is returned at `+1`, and the caller is responsible for freeing it. Similarly, the annotation `__attribute__((ns_returns_not_retained))` specifies that the object is returned at `+0` and the ownership remains with the callee. The annotation `__attribute__((ns_consumes_self))` specifies that the Objective-C method call consumes the reference to `self`, e.g. by attaching it to a supplied parameter. Additionally, parameters

can have an annotation `__attribute__((ns_consumed))`, which specifies that passing an owned object as that parameter effectively transfers the ownership, and the caller is no longer responsible for it. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

In C programs using CoreFoundation, a similar set of attributes: `__attribute__((cf_returns_not_retained))`, `__attribute__((cf_returns_retained))` and `__attribute__((cf_consumed))` have the same respective semantics when applied to CoreFoundation objects. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

Finally, in C++ interacting with XNU kernel (objects inheriting from OSObject), the same attribute family is present: `__attribute__((os_returns_not_retained))`, `__attribute__((os_returns_retained))` and `__attribute__((os_consumed))`, with the same respective semantics. Similar to `__attribute__((ns_consumes_self))`, `__attribute__((os_consumes_this))` specifies that the method call consumes the reference to "this" (e.g., when attaching it to a different object supplied as a parameter). Out parameters (parameters the function is meant to write into, either via pointers-to-pointers or references-to-pointers) may be annotated with `__attribute__((os_returns_retained))` or `__attribute__((os_returns_not_retained))` which specifies that the object written into the out parameter should (or respectively should not) be released after use. Since often out parameters may or may not be written depending on the exit code of the function, annotations `__attribute__((os_returns_retained_on_zero))` and `__attribute__((os_returns_retained_on_non_zero))` specify that an out parameter at +1 is written if and only if the function returns a zero (respectively non-zero) error code. Observe that return-code-dependent out parameter annotations are only available for retained out parameters, as non-retained object do not have to be released by the callee. These attributes are only used by the Clang Static Analyzer.

The family of attributes `X_returns_X_retained` can be added to functions, C++ methods, and Objective-C methods and properties. Attributes `X_consumed` can be added to parameters of methods, functions, and Objective-C methods.

cfi_canonical_jump_table

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
cfi_canonical_jump_table	clang::cfi_canonical_jump_table	clang::cfi_canonical_jump_table					Yes

Use `__attribute__((cfi_canonical_jump_table))` on a function declaration to make the function's CFI jump table canonical. See the CFI documentation for more details.

clang::builtin_alias, clang_builtin_alias

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
clang_builtin_alias	clang::builtin_alias	clang::builtin_alias					Yes

This attribute is used in the implementation of the C intrinsics. It allows the C intrinsic functions to be declared using the names defined in target builtins, and still be recognized as clang builtins equivalent to the underlying name. For example, `riscv_vector.h` declares the function `vadd` with `__attribute__((clang_builtin_alias(__builtin_rvv_vadd_vv_i8m1)))`. This ensures that both

functions are recognized as that clang builtin, and in the latter case, the choice of which builtin to identify the function as can be deferred until after overload resolution.

This attribute can only be used to set up the aliases for certain ARM/RISC-V C intrinsic functions; it is intended for use only inside `arm_*.h` and `riscv_*.h` and is not a general mechanism for declaring arbitrary aliases for clang builtin functions.

clang_arm_builtin_alias

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
`__clang_arm_builtin_alias`	`clang::__clang_arm_builtin_alias`	`clang::__clang_arm_builtin_alias`					Yes

This attribute is used in the implementation of the ACLE intrinsics. It allows the intrinsic functions to be declared using the names defined in ACLE, and still be recognized as clang builtins equivalent to the underlying name. For example, `arm_mve.h` declares the function `vaddq_u32` with `__attribute__((__clang_arm_mve_alias(__builtin_arm_mve_vaddq_u32)))`, and similarly, one of the type-overloaded declarations of `vaddq` will have the same attribute. This ensures that both functions are recognized as that clang builtin, and in the latter case, the choice of which builtin to identify the function as can be deferred until after overload resolution.

This attribute can only be used to set up the aliases for certain Arm intrinsic functions; it is intended for use only inside `arm_*.h` and is not a general mechanism for declaring arbitrary aliases for clang builtin functions.

In order to avoid duplicating the attribute definitions for similar purpose for other architecture, there is a general form for the attribute *clang_builtin_alias*.

cmse_nonsecure_entry

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
`cmse_nonsecure_entry`							Yes

This attribute declares a function that can be called from non-secure state, or from secure state. Entering from and returning to non-secure state would switch to and from secure state, respectively, and prevent flow of information to non-secure state, except via return values. See ARMv8-M Security Extensions: Requirements on Development Tools - Engineering Specification Documentation for more information.

code_seg

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
			`code_seg`				

The `__declspec(code_seg)` attribute enables the placement of code into separate named segments that can be paged or locked in memory individually. This attribute is used to control the placement of instantiated templates and compiler-generated code. See the documentation for __declspec(code_seg) on MSDN.

convergent

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
convergent	clang::convergent	clang::convergent					Yes

The `convergent` attribute can be placed on a function declaration. It is translated into the LLVM `convergent` attribute, which indicates that the call instructions of a function with this attribute cannot be made control-dependent on any additional values.

In languages designed for SPMD/SIMT programming model, e.g. OpenCL or CUDA, the call instructions of a function with this attribute must be executed by all work items or threads in a work group or sub group.

This attribute is different from `noduplicate` because it allows duplicating function calls if it can be proved that the duplicated function calls are not made control-dependent on any additional values, e.g., unrolling a loop executed by all work items.

Sample usage:

```
void convfunc(void) __attribute__((convergent));
// Setting it as a C++11 attribute is also valid in a C++ program.
// void convfunc(void) [[clang::convergent]];
```

cpu_dispatch

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
cpu_dispatch	clang::cpu_dispatch	clang::cpu_dispatch	cpu_dispatch				Yes

The `cpu_specific` and `cpu_dispatch` attributes are used to define and resolve multiversioned functions. This form of multiversioning provides a mechanism for declaring versions across translation units and manually specifying the resolved function list. A specified CPU defines a set of minimum features that are required for the function to be called. The result of this is that future processors execute the most restrictive version of the function the new processor can execute.

In addition, unlike the ICC implementation of this feature, the selection of the version does not consider the manufacturer or microarchitecture of the processor. It tests solely the list of features that are both supported by the specified processor and present in the compiler-rt library. This can be surprising at times, as the runtime processor may be from a completely different manufacturer, as long as it supports the same feature set.

This can additionally be surprising, as some processors are indistinguishable from others based on the list of testable features. When this happens, the variant is selected in an unspecified manner.

Function versions are defined with `cpu_specific`, which takes one or more CPU names as a parameter. For example:

```
// Declares and defines the ivybridge version of single_cpu.
__attribute__((cpu_specific(ivybridge)))
void single_cpu(void){}

// Declares and defines the atom version of single_cpu.
__attribute__((cpu_specific(atom)))
void single_cpu(void){}

// Declares and defines both the ivybridge and atom version of multi_cpu.
__attribute__((cpu_specific(ivybridge, atom)))
void multi_cpu(void){}
```

A dispatching (or resolving) function can be declared anywhere in a project's source code with `cpu_dispatch`. This attribute takes one or more CPU names as a parameter (like `cpu_specific`). Functions marked with `cpu_dispatch` are not expected to be defined, only declared. If such a marked function has a definition, any side effects of the function are ignored; trivial function bodies are permissible for ICC compatibility.

```
// Creates a resolver for single_cpu above.
__attribute__((cpu_dispatch(ivybridge, atom)))
void single_cpu(void){}

// Creates a resolver for multi_cpu, but adds a 3rd version defined in another
// translation unit.
__attribute__((cpu_dispatch(ivybridge, atom, sandybridge)))
void multi_cpu(void){}
```

Note that it is possible to have a resolving function that dispatches based on more or fewer options than are present in the program. Specifying fewer will result in the omitted options not being considered during resolution. Specifying a version for resolution that isn't defined in the program will result in a linking failure.

It is also possible to specify a CPU name of `generic` which will be resolved if the executing processor doesn't satisfy the features required in the CPU name. The behavior of a program executing on a processor that doesn't satisfy any option of a multiversioned function is undefined.

cpu_specific

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
cpu_specific	clang::cpu_specific	clang::cpu_specific	cpu_specific				Yes

The `cpu_specific` and `cpu_dispatch` attributes are used to define and resolve multiversioned functions. This form of multiversioning provides a mechanism for declaring versions across translation units and manually specifying the resolved function list. A specified CPU defines a set of minimum features that are required for the function to be called. The result of this is that future processors execute the most restrictive version of the function the new processor can execute.

In addition, unlike the ICC implementation of this feature, the selection of the version does not consider the manufacturer or microarchitecture of the processor. It tests solely the list of features that are both supported by the specified processor and present in the compiler-rt library. This can be surprising at times, as the runtime processor may be from a completely different manufacturer, as long as it supports the same feature set.

This can additionally be surprising, as some processors are indistringuishable from others based on the list of testable features. When this happens, the variant is selected in an unspecified manner.

Function versions are defined with `cpu_specific`, which takes one or more CPU names as a parameter. For example:

```
// Declares and defines the ivybridge version of single_cpu.
__attribute__((cpu_specific(ivybridge)))
void single_cpu(void){}

// Declares and defines the atom version of single_cpu.
__attribute__((cpu_specific(atom)))
void single_cpu(void){}

// Declares and defines both the ivybridge and atom version of multi_cpu.
__attribute__((cpu_specific(ivybridge, atom)))
void multi_cpu(void){}
```

A dispatching (or resolving) function can be declared anywhere in a project's source code with `cpu_dispatch`. This attribute takes one or more CPU names as a parameter (like `cpu_specific`). Functions marked with `cpu_dispatch` are not expected to be defined, only declared. If such a marked function has a definition, any side effects of the function are ignored; trivial function bodies are permissible for ICC compatibility.

```
// Creates a resolver for single_cpu above.
__attribute__((cpu_dispatch(ivybridge, atom)))
void single_cpu(void){}

// Creates a resolver for multi_cpu, but adds a 3rd version defined in another
// translation unit.
__attribute__((cpu_dispatch(ivybridge, atom, sandybridge)))
void multi_cpu(void){}
```

Note that it is possible to have a resolving function that dispatches based on more or fewer options than are present in the program. Specifying fewer will result in the omitted options not being considered during resolution. Specifying a version for resolution that isn't defined in the program will result in a linking failure.

It is also possible to specify a CPU name of `generic` which will be resolved if the executing processor doesn't satisfy the features required in the CPU name. The behavior of a program executing on a processor that doesn't satisfy any option of a multiversioned function is undefined.

diagnose_as_builtin

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
diagnose_as_builtin	clang::diagnose_as_builtin	clang::diagnose_as_builtin					Yes

The `diagnose_as_builtin` attribute indicates that Fortify diagnostics are to be applied to the declared function as if it were the function specified by the attribute. The builtin function whose diagnostics are to be mimicked should be given. In addition, the order in which arguments should be applied must also be given.

For example, the attribute can be used as follows.

```
__attribute__((diagnose_as_builtin(__builtin_memset, 3, 2, 1)))
void *mymemset(int n, int c, void *s) {
  // ...
}
```

This indicates that calls to `mymemset` should be diagnosed as if they were calls to `__builtin_memset`. The arguments `3, 2, 1` indicate by index the order in which arguments of `mymemset` should be applied to `__builtin_memset`. The third argument should be applied first, then the second, and then the first. Thus (when

Fortify warnings are enabled) the call `mymemset(n, c, s)` will diagnose overflows as if it were the call `__builtin_memset(s, c, n)`.

For variadic functions, the variadic arguments must come in the same order as they would to the builtin function, after all normal arguments. For instance, to diagnose a new function as if it were *sscanf*, we can use the attribute as follows.

```
__attribute__((diagnose_as_builtin(sscanf, 1, 2)))
int mysscanf(const char *str, const char *format, ...)  {
  // ...
}
```

Then the call *mysscanf("abc def", "%4s %4s", buf1, buf2)* will be diagnosed as if it were the call *sscanf("abc def", "%4s %4s", buf1, buf2)*.

This attribute cannot be applied to non-static member functions.

diagnose_if

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
diagnose_if							

The `diagnose_if` attribute can be placed on function declarations to emit warnings or errors at compile-time if calls to the attributed function meet certain user-defined criteria. For example:

```
int abs(int a)
  __attribute__((diagnose_if(a >= 0, "Redundant abs call", "warning")));
int must_abs(int a)
  __attribute__((diagnose_if(a >= 0, "Redundant abs call", "error")));

int val = abs(1); // warning: Redundant abs call
int val2 = must_abs(1); // error: Redundant abs call
int val3 = abs(val);
int val4 = must_abs(val); // Because run-time checks are not emitted for
                          // diagnose_if attributes, this executes without
                          // issue.
```

`diagnose_if` is closely related to `enable_if`, with a few key differences:

- Overload resolution is not aware of `diagnose_if` attributes: they're considered only after we select the best candidate from a given candidate set.
- Function declarations that differ only in their `diagnose_if` attributes are considered to be redeclarations of the same function (not overloads).
- If the condition provided to `diagnose_if` cannot be evaluated, no diagnostic will be emitted.

Otherwise, `diagnose_if` is essentially the logical negation of `enable_if`.

As a result of bullet number two, `diagnose_if` attributes will stack on the same function. For example:

```
int foo() __attribute__((diagnose_if(1, "diag1", "warning")));
int foo() __attribute__((diagnose_if(1, "diag2", "warning")));

int bar = foo(); // warning: diag1
                 // warning: diag2
int (*fooptr)(void) = foo; // warning: diag1
                           // warning: diag2
```

```
constexpr int supportsAPILevel(int N) { return N < 5; }
int baz(int a)
  __attribute__((diagnose_if(!supportsAPILevel(10),
                             "Upgrade to API level 10 to use baz", "error")));
int baz(int a)
  __attribute__((diagnose_if(!a, "0 is not recommended.", "warning")));

int (*bazptr)(int) = baz; // error: Upgrade to API level 10 to use baz
int v = baz(0); // error: Upgrade to API level 10 to use baz
```

Query for this feature with `__has_attribute(diagnose_if)`.

disable_sanitizer_instrumentation

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
disable_sanitizer_instrumentation	clang::disable_sanitizer_instrumentation	clang::disable_sanitizer_instrumentation					Yes

Use the `disable_sanitizer_instrumentation` attribute on a function, Objective-C method, or global variable, to specify that no sanitizer instrumentation should be applied.

This is not the same as `__attribute__((no_sanitize(...)))`, which depending on the tool may still insert instrumentation to prevent false positive reports.

disable_tail_calls

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
disable_tail_calls	clang::disable_tail_calls	clang::disable_tail_calls					Yes

The `disable_tail_calls` attribute instructs the backend to not perform tail call optimization inside the marked function.

For example:

```
int callee(int);

int foo(int a) __attribute__((disable_tail_calls)) {
  return callee(a); // This call is not tail-call optimized.
}
```

Marking virtual functions as `disable_tail_calls` is legal.

```
int callee(int);
```

```
class Base {
public:
  [[clang::disable_tail_calls]] virtual int foo1() {
    return callee(); // This call is not tail-call optimized.
  }
};

class Derived1 : public Base {
public:
  int foo1() override {
    return callee(); // This call is tail-call optimized.
  }
};
```

enable_if

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
enable_if							Yes

Note

Some features of this attribute are experimental. The meaning of multiple enable_if attributes on a single declaration is subject to change in a future version of clang. Also, the ABI is not standardized and the name mangling may change in future versions. To avoid that, use asm labels.

The `enable_if` attribute can be placed on function declarations to control which overload is selected based on the values of the function's arguments. When combined with the `overloadable` attribute, this feature is also available in C.

```
int isdigit(int c);
int isdigit(int c) __attribute__((enable_if(c <= -1 || c > 255, "chosen when 'c' is out of range"))) __attribute__((unavailable("'c' must have the value of an unsigned char or EOF")));

void foo(char c) {
  isdigit(c);
  isdigit(10);
  isdigit(-10);  // results in a compile-time error.
}
```

The enable_if attribute takes two arguments, the first is an expression written in terms of the function parameters, the second is a string explaining why this overload candidate could not be selected to be displayed in diagnostics. The expression is part of the function signature for the purposes of determining whether it is a redeclaration (following the rules used when determining whether a C++ template specialization is ODR-equivalent), but is not part of the type.

The enable_if expression is evaluated as if it were the body of a bool-returning constexpr function declared with the arguments of the function it is being applied to, then called with the parameters at the call site. If the result is false or could not be determined through constant expression evaluation, then this overload will not be chosen and the provided string may be used in a diagnostic if the compile fails as a result.

Because the enable_if expression is an unevaluated context, there are no global state changes, nor the ability to pass information from the enable_if expression to the function body. For example, suppose we want calls to strnlen(strbuf, maxlen) to resolve to strnlen_chk(strbuf, maxlen, size of strbuf) only if the size of strbuf can be determined:

```
__attribute__((always_inline))
static inline size_t strnlen(const char *s, size_t maxlen)
  __attribute__((overloadable))
  __attribute__((enable_if(__builtin_object_size(s, 0) != -1))),
                        "chosen when the buffer size is known but 'maxlen' is not")))
```

```
{
  return strnlen_chk(s, maxlen, __builtin_object_size(s, 0));
}
```

Multiple enable_if attributes may be applied to a single declaration. In this case, the enable_if expressions are evaluated from left to right in the following manner. First, the candidates whose enable_if expressions evaluate to false or cannot be evaluated are discarded. If the remaining candidates do not share ODR-equivalent enable_if expressions, the overload resolution is ambiguous. Otherwise, enable_if overload resolution continues with the next enable_if attribute on the candidates that have not been discarded and have remaining enable_if attributes. In this way, we pick the most specific overload out of a number of viable overloads using enable_if.

```
void f() __attribute__((enable_if(true, "")));  // #1
void f() __attribute__((enable_if(true, ""))) __attribute__((enable_if(true, "")));  // #2

void g(int i, int j) __attribute__((enable_if(i, "")));  // #1
void g(int i, int j) __attribute__((enable_if(j, ""))) __attribute__((enable_if(true)));  // #2
```

In this example, a call to f() is always resolved to #2, as the first enable_if expression is ODR-equivalent for both declarations, but #1 does not have another enable_if expression to continue evaluating, so the next round of evaluation has only a single candidate. In a call to g(1, 1), the call is ambiguous even though #2 has more enable_if attributes, because the first enable_if expressions are not ODR-equivalent.

Query for this feature with `__has_attribute(enable_if)`.

Note that functions with one or more `enable_if` attributes may not have their address taken, unless all of the conditions specified by said `enable_if` are constants that evaluate to `true`. For example:

```
const int TrueConstant = 1;
const int FalseConstant = 0;
int f(int a) __attribute__((enable_if(a > 0, "")));
int g(int a) __attribute__((enable_if(a == 0 || a != 0, "")));
int h(int a) __attribute__((enable_if(1, "")));
int i(int a) __attribute__((enable_if(TrueConstant, "")));
int j(int a) __attribute__((enable_if(FalseConstant, "")));

void fn() {
  int (*ptr)(int);
  ptr = &f; // error: 'a > 0' is not always true
  ptr = &g; // error: 'a == 0 || a != 0' is not a truthy constant
  ptr = &h; // OK: 1 is a truthy constant
  ptr = &i; // OK: 'TrueConstant' is a truthy constant
  ptr = &j; // error: 'FalseConstant' is a constant, but not truthy
}
```

Because `enable_if` evaluation happens during overload resolution, `enable_if` may give unintuitive results when used with templates, depending on when overloads are resolved. In the example below, clang will emit a diagnostic about no viable overloads for `foo` in `bar`, but not in `baz`:

```
double foo(int i) __attribute__((enable_if(i > 0, "")));
void *foo(int i) __attribute__((enable_if(i <= 0, "")));
template <int I>
auto bar() { return foo(I); }

template <typename T>
auto baz() { return foo(T::number); }

struct WithNumber { constexpr static int number = 1; };
void callThem() {
  bar<sizeof(WithNumber)>();
  baz<WithNumber>();
}
```

This is because, in `bar`, `foo` is resolved prior to template instantiation, so the value for `I` isn't known (thus, both `enable_if` conditions for `foo` fail). However, in `baz`, `foo` is resolved during template instantiation, so the value for `T::number` is known.

enforce_tcb

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
enforce_tcb	clang::enforce_tcb	clang::enforce_tcb					Yes

The `enforce_tcb` attribute can be placed on functions to enforce that a

trusted compute base (TCB) does not call out of the TCB. This generates a warning every time a function not marked with an `enforce_tcb` attribute is called from a function with the `enforce_tcb` attribute. A function may be a part of multiple TCBs. Invocations through function pointers are currently not checked. Builtins are considered to a part of every TCB.

- `enforce_tcb(Name)` indicates that this function is a part of the TCB named `Name`

enforce_tcb_leaf

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
enforce_tcb_leaf	clang::enforce_tcb_leaf	clang::enforce_tcb_leaf					Yes

The `enforce_tcb_leaf` attribute satisfies the requirement enforced by

`enforce_tcb` for the marked function to be in the named TCB but does not continue to check the functions called from within the leaf function.

- `enforce_tcb_leaf(Name)` indicates that this function is a part of the TCB named `Name`

error, warning

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
error warning	gnu::error gnu::warning	gnu::error gnu::warning					Yes

The `error` and `warning` function attributes can be used to specify a custom diagnostic to be emitted when a call to such a function is not eliminated via optimizations. This can be used to create compile time assertions that depend on optimizations, while providing diagnostics pointing to precise locations of the call site in the source.

```
__attribute__((warning("oh no"))) void dontcall();
void foo() {
  if (someCompileTimeAssertionThatsTrue)
    dontcall(); // Warning

  dontcall(); // Warning

  if (someCompileTimeAssertionThatsFalse)
    dontcall(); // No Warning
  sizeof(dontcall()); // No Warning
}
```

exclude_from_explicit_instantiation

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
exclude_from_explicit_instantiation	clang::exclude_from_explicit_instantiation	clang::exclude_from_explicit_instantiation					Yes

The `exclude_from_explicit_instantiation` attribute opts-out a member of a class template from being part of explicit template instantiations of that class template. This means that an explicit instantiation will not instantiate members of the class template marked with the attribute, but also that code where an extern template declaration of the enclosing class template is visible will not take for granted that an external instantiation of the class template would provide those members (which would otherwise be a link error, since the explicit instantiation won't provide those members). For example, let's say we don't want the `data()` method to be part of libc++'s ABI. To make sure it is not exported from the dylib, we give it hidden visibility:

```
// in <string>
template <class CharT>
class basic_string {
public:
  __attribute__((__visibility__("hidden")))
  const value_type* data() const noexcept { ... }
};

template class basic_string<char>;
```

Since an explicit template instantiation declaration for `basic_string<char>` is provided, the compiler is free to assume that `basic_string<char>::data()` will be provided by another translation unit, and it is free to produce an external call to this function. However, since `data()` has hidden visibility and the explicit template instantiation is provided in a shared library (as opposed to simply another translation unit), `basic_string<char>::data()` won't be found and a link error will ensue. This happens because the compiler assumes that `basic_string<char>::data()` is part of the explicit template instantiation declaration, when it really isn't. To tell the compiler that `data()` is not part of the explicit template instantiation declaration, the `exclude_from_explicit_instantiation` attribute can be used:

```
// in <string>
template <class CharT>
class basic_string {
public:
  __attribute__((__visibility__("hidden")))
```

```
  __attribute__((exclude_from_explicit_instantiation))
  const value_type* data() const noexcept { ... }
};

template class basic_string<char>;
```

Now, the compiler won't assume that `basic_string<char>::data()` is provided externally despite there being an explicit template instantiation declaration: the compiler will implicitly instantiate `basic_string<char>::data()` in the TUs where it is used.

This attribute can be used on static and non-static member functions of class templates, static data members of class templates and member classes of class templates.

export_name

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
export_name	clang::export_name	clang::export_name					Yes

Clang supports the `__attribute__((export_name(<name>)))` attribute for the WebAssembly target. This attribute may be attached to a function declaration, where it modifies how the symbol is to be exported from the linked WebAssembly.

WebAssembly functions are exported via string name. By default when a symbol is exported, the export name for C/C++ symbols are the same as their C/C++ symbol names. This attribute can be used to override the default behavior, and request a specific string name be used instead.

flatten

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
flatten	gnu::flatten	gnu::flatten					Yes

The `flatten` attribute causes calls within the attributed function to be inlined unless it is impossible to do so, for example if the body of the callee is unavailable or if the callee has the `noinline` attribute.

force_align_arg_pointer

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
force_align_arg_pointer	gnu::force_align_arg_pointer	gnu::force_align_arg_pointer					

Use this attribute to force stack alignment.

Legacy x86 code uses 4-byte stack alignment. Newer aligned SSE instructions (like ‘movaps’) that work with the stack require operands to be 16-byte aligned. This attribute realigns the stack in the function prologue to make sure the stack can be used with SSE instructions.

Note that the x86_64 ABI forces 16-byte stack alignment at the call site. Because of this, ‘force_align_arg_pointer’ is not needed on x86_64, except in rare cases where the caller does not align the stack properly (e.g. flow jumps from i386 arch code).

```
__attribute__ ((force_align_arg_pointer))
void f () {
  ...
}
```

format

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
format	gnu::format	gnu::format					

Clang supports the `format` attribute, which indicates that the function accepts a `printf` or `scanf`-like format string and corresponding arguments or a `va_list` that contains these arguments.

Please see GCC documentation about format attribute to find details about attribute syntax.

Clang implements two kinds of checks with this attribute.

1. Clang checks that the function with the `format` attribute is called with a format string that uses format specifiers that are allowed, and that arguments match the format string. This is the `-Wformat` warning, it is on by default.
2. Clang checks that the format string argument is a literal string. This is the `-Wformat-nonliteral` warning, it is off by default.

   Clang implements this mostly the same way as GCC, but there is a difference for functions that accept a `va_list` argument (for example, `vprintf`). GCC does not emit `-Wformat-nonliteral` warning for calls to such functions. Clang does not warn if the format string comes from a function parameter, where the function is annotated with a compatible attribute, otherwise it warns. For example:

   ```
   __attribute__((__format__ (__scanf__, 1, 3)))
   void foo(const char* s, char *buf, ...) {
     va_list ap;
     va_start(ap, buf);

     vprintf(s, ap); // warning: format string is not a string literal
   }
   ```

   In this case we warn because `s` contains a format string for a `scanf`-like function, but it is passed to a `printf`-like function.

   If the attribute is removed, clang still warns, because the format string is not a string literal.

   Another example:

   ```
   __attribute__((__format__ (__printf__, 1, 3)))
   void foo(const char* s, char *buf, ...) {
     va_list ap;
     va_start(ap, buf);
   ```

```
  vprintf(s, ap); // warning
}
```

In this case Clang does not warn because the format string `s` and the corresponding arguments are annotated. If the arguments are incorrect, the caller of `foo` will receive a warning.

gnu_inline

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
gnu_inline	gnu::gnu_inline	gnu::gnu_inline					Yes

The `gnu_inline` changes the meaning of `extern inline` to use GNU inline semantics, meaning:

- If any declaration that is declared `inline` is not declared `extern`, then the `inline` keyword is just a hint. In particular, an out-of-line definition is still emitted for a function with external linkage, even if all call sites are inlined, unlike in C99 and C++ inline semantics.
- If all declarations that are declared `inline` are also declared `extern`, then the function body is present only for inlining and no out-of-line version is emitted.

Some important consequences: `static inline` emits an out-of-line version if needed, a plain `inline` definition emits an out-of-line version always, and an `extern inline` definition (in a header) followed by a (non-`extern`) `inline` declaration in a source file emits an out-of-line version of the function in that source file but provides the function body for inlining to all includers of the header.

Either `__GNUC_GNU_INLINE__` (GNU inline semantics) or `__GNUC_STDC_INLINE__` (C99 semantics) will be defined (they are mutually exclusive). If `__GNUC_STDC_INLINE__` is defined, then the `gnu_inline` function attribute can be used to get GNU inline semantics on a per function basis. If `__GNUC_GNU_INLINE__` is defined, then the translation unit is already being compiled with GNU inline semantics as the implied default. It is unspecified which macro is defined in a C++ compilation.

GNU inline semantics are the default behavior with `-std=gnu89`, `-std=c89`, `-std=c94`, or `-fgnu89-inline`.

guard

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
			guard				

Code can indicate CFG checks are not wanted with the `__declspec(guard(nocf))` attribute. This directs the compiler to not insert any CFG checks for the entire function. This approach is typically used only sparingly in specific situations where the programmer has manually inserted “CFG-equivalent” protection. The programmer knows that they are calling through some read-only function table whose address is obtained through read-only memory references and for which the index is masked to the function table limit. This approach may also be applied to small wrapper functions that are not inlined and that do nothing more than make a call through a function pointer. Since incorrect usage of this directive can compromise the security of CFG, the programmer must be very careful using the directive. Typically, this usage is limited to very small functions that only call one function.

Control Flow Guard documentation <https://docs.microsoft.com/en-us/windows/win32/secbp/pe-metadata>

ifunc

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
ifunc	gnu::ifunc	gnu::ifunc					Yes

`__attribute__((ifunc("resolver")))` is used to mark that the address of a declaration should be resolved at runtime by calling a resolver function.

The symbol name of the resolver function is given in quotes. A function with this name (after mangling) must be defined in the current translation unit; it may be `static`. The resolver function should return a pointer.

The `ifunc` attribute may only be used on a function declaration. A function declaration with an `ifunc` attribute is considered to be a definition of the declared entity. The entity must not have weak linkage; for example, in C++, it cannot be applied to a declaration if a definition at that location would be considered inline.

Not all targets support this attribute. ELF target support depends on both the linker and runtime linker, and is available in at least lld 4.0 and later, binutils 2.20.1 and later, glibc v2.11.1 and later, and FreeBSD 9.1 and later. Non-ELF targets currently do not support this attribute.

import_module

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
import_module	clang::import_module	clang::import_module					Yes

Clang supports the `__attribute__((import_module(<module_name>)))` attribute for the WebAssembly target. This attribute may be attached to a function declaration, where it modifies how the symbol is to be imported within the WebAssembly linking environment.

WebAssembly imports use a two-level namespace scheme, consisting of a module name, which typically identifies a module from which to import, and a field name, which typically identifies a field from that module to import. By default, module names for C/C++ symbols are assigned automatically by the linker. This attribute can be used to override the default behavior, and request a specific module name be used instead.

import_name

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
import_name	clang::import_name	clang::import_name					Yes

Clang supports the `__attribute__((import_name(<name>)))` attribute for the WebAssembly target. This attribute may be attached to a function declaration, where it modifies how the symbol is to be imported within the WebAssembly linking environment.

WebAssembly imports use a two-level namespace scheme, consisting of a module name, which typically identifies a module from which to import, and a field name, which typically identifies a field from that module to import. By default, field names for C/C++ symbols are the same as their C/C++ symbol names. This attribute can be used to override the default behavior, and request a specific field name be used instead.

internal_linkage

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
internal_linkage	clang::internal_linkage	clang::internal_linkage					Yes

The `internal_linkage` attribute changes the linkage type of the declaration to internal. This is similar to C-style `static`, but can be used on classes and class methods. When applied to a class definition, this attribute affects all methods and static data members of that class. This can be used to contain the ABI of a C++ library by excluding unwanted class methods from the export tables.

interrupt (ARM)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
interrupt	gnu::interrupt	gnu::interrupt					

Clang supports the GNU style `__attribute__((interrupt("TYPE")))` attribute on ARM targets. This attribute may be attached to a function definition and instructs the backend to generate appropriate function entry/exit code so that it can be used directly as an interrupt service routine.

The parameter passed to the interrupt attribute is optional, but if provided it must be a string literal with one of the following values: “IRQ”, “FIQ”, “SWI”, “ABORT”, “UNDEF”.

The semantics are as follows:

- If the function is AAPCS, Clang instructs the backend to realign the stack to 8 bytes on entry. This is a general requirement of the AAPCS at public interfaces, but may not hold when an exception is taken. Doing this allows other AAPCS functions to be called.
- If the CPU is M-class this is all that needs to be done since the architecture itself is designed in such a way that functions obeying the normal AAPCS ABI constraints are valid exception handlers.
- If the CPU is not M-class, the prologue and epilogue are modified to save all non-banked registers that are used, so that upon return the user-mode state will not be corrupted. Note that to avoid unnecessary overhead, only general-purpose (integer) registers are saved in this way. If VFP operations are needed, that state must be saved manually.

  Specifically, interrupt kinds other than “FIQ” will save all core registers except “lr” and “sp”. “FIQ” interrupts will save r0-r7.
- If the CPU is not M-class, the return instruction is changed to one of the canonical sequences permitted by the architecture for exception return. Where possible the function itself will make the necessary “lr” adjustments so that the “preferred return address” is selected.
  Unfortunately the compiler is unable to make this guarantee for an “UNDEF” handler, where the offset from “lr” to the preferred return address depends on the execution state of the code which generated the exception. In this case a sequence equivalent to “movs pc, lr” will be used.

interrupt (AVR)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
interrupt	gnu::interrupt	gnu::interrupt					Yes

Clang supports the GNU style `__attribute__((interrupt))` attribute on AVR targets. This attribute may be attached to a function definition and instructs the backend to generate appropriate function entry/exit code so that it can be used directly as an interrupt service routine.

On the AVR, the hardware globally disables interrupts when an interrupt is executed. The first instruction of an interrupt handler declared with this attribute is a SEI instruction to re-enable interrupts. See also the signal attribute that does not insert a SEI instruction.

interrupt (MIPS)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
interrupt	gnu::interrupt	gnu::interrupt					Yes

Clang supports the GNU style `__attribute__((interrupt("ARGUMENT")))` attribute on MIPS targets. This attribute may be attached to a function definition and instructs the backend to generate appropriate function entry/exit code so that it can be used directly as an interrupt service routine.

By default, the compiler will produce a function prologue and epilogue suitable for an interrupt service routine that handles an External Interrupt Controller (eic) generated interrupt. This behavior can be explicitly requested with the “eic” argument.

Otherwise, for use with vectored interrupt mode, the argument passed should be of the form “vector=LEVEL” where LEVEL is one of the following values: “sw0”, “sw1”, “hw0”, “hw1”, “hw2”, “hw3”, “hw4”, “hw5”. The compiler will then set the interrupt mask to the corresponding level which will mask all interrupts up to and including the argument.

The semantics are as follows:

- The prologue is modified so that the Exception Program Counter (EPC) and Status coprocessor registers are saved to the stack. The interrupt mask is set so that the function can only be interrupted by a higher priority interrupt. The epilogue will restore the previous values of EPC and Status.
- The prologue and epilogue are modified to save and restore all non-kernel registers as necessary.
- The FPU is disabled in the prologue, as the floating pointer registers are not spilled to the stack.
- The function return sequence is changed to use an exception return instruction.
- The parameter sets the interrupt mask for the function corresponding to the interrupt level specified. If no mask is specified the interrupt mask defaults to “eic”.

interrupt (RISCV)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
interrupt	gnu::interrupt	gnu::interrupt					Yes

Clang supports the GNU style `__attribute__((interrupt))` attribute on RISCV targets. This attribute may be attached to a function definition and instructs the backend to generate appropriate function entry/exit code so that it can be used directly as an interrupt service routine.

Permissible values for this parameter are `user`, `supervisor`, and `machine`. If there is no parameter, then it defaults to machine.

Repeated interrupt attribute on the same declaration will cause a warning to be emitted. In case of repeated declarations, the last one prevails.

Refer to: https://gcc.gnu.org/onlinedocs/gcc/RISC-V-Function-Attributes.html https://riscv.org/specifications/privileged-isa/ The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10.

kernel

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
kernel							Yes

`__attribute__((kernel))` is used to mark a `kernel` function in RenderScript.

In RenderScript, `kernel` functions are used to express data-parallel computations. The RenderScript runtime efficiently parallelizes `kernel` functions to run on computational resources such as multi-core CPUs and GPUs. See the RenderScript documentation for more information.

lifetimebound

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
lifetimebound	clang::lifetimebound						

The `lifetimebound` attribute on a function parameter or implicit object parameter indicates that objects that are referred to by that parameter may also be referred to by the return value of the annotated function (or, for a parameter of a constructor, by the value of the constructed object). It is only supported in C++.

By default, a reference is considered to refer to its referenced object, a pointer is considered to refer to its pointee, a `std::initializer_list<T>` is considered to refer to its underlying array, and aggregates (arrays and simple `structs`) are considered to refer to all objects that their transitive subobjects refer to.

Clang warns if it is able to detect that an object or reference refers to another object with a shorter lifetime. For example, Clang will warn if a function returns a reference to a local variable, or if a reference is bound to a temporary object whose lifetime is not extended. By using the `lifetimebound` attribute, this determination can be extended to look through user-declared functions. For example:

```
// Returns m[key] if key is present, or default_value if not.
template<typename T, typename U>
```

```
const U &get_or_default(const std::map<T, U> &m [[clang::lifetimebound]],
                        const T &key, /* note, not lifetimebound */
                        const U &default_value [[clang::lifetimebound]]);

std::map<std::string, std::string> m;
// warning: temporary "bar"s that might be bound to local reference 'val'
// will be destroyed at the end of the full-expression
const std::string &val = get_or_default(m, "foo"s, "bar"s);

// No warning in this case.
std::string def_val = "bar"s;
const std::string &val = get_or_default(m, "foo"s, def_val);
```

The attribute can be applied to the implicit `this` parameter of a member function by writing the attribute after the function type:

```
struct string {
  // The returned pointer should not outlive ``*this``.
  const char *data() const [[clang::lifetimebound]];
};
```

This attribute is inspired by the C++ committee paper P0936R0, but does not affect whether temporary objects have their lifetimes extended.

long_call, far

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
long_call far	gnu::long_call gnu::far	gnu::long_call gnu::far					Yes

Clang supports the `__attribute__((long_call))`, `__attribute__((far))`, and `__attribute__((near))` attributes on MIPS targets. These attributes may only be added to function declarations and change the code generated by the compiler when directly calling the function. The `near` attribute allows calls to the function to be made using the `jal` instruction, which requires the function to be located in the same naturally aligned 256MB segment as the caller. The `long_call` and `far` attributes are synonyms and require the use of a different call sequence that works regardless of the distance between the functions.

These attributes have no effect for position-independent code.

These attributes take priority over command line switches such as `-mlong-calls` and `-mno-long-calls`.

malloc

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
malloc	gnu::malloc	gnu::malloc	restrict				Yes

The `malloc` attribute indicates that the function acts like a system memory allocation function, returning a pointer to allocated storage disjoint from the storage for any other object accessible to the caller.

micromips

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
micromips	gnu::micromips	gnu::micromips					Yes

Clang supports the GNU style `__attribute__((micromips))` and `__attribute__((nomicromips))` attributes on MIPS targets. These attributes may be attached to a function definition and instructs the backend to generate or not to generate microMIPS code for that function.

These attributes override the `-mmicromips` and `-mno-micromips` options on the command line.

mig_server_routine

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
mig_server_routine	clang::mig_server_routine	clang::mig_server_routine					Yes

The Mach Interface Generator release-on-success convention dictates functions that follow it to only release arguments passed to them when they return “success” (a `kern_return_t` error code that indicates that no errors have occurred). Otherwise the release is performed by the MIG client that called the function. The annotation `__attribute__((mig_server_routine))` is applied in order to specify which functions are expected to follow the convention. This allows the Static Analyzer to find bugs caused by violations of that convention. The attribute would normally appear on the forward declaration of the actual server routine in the MIG server header, but it may also be added to arbitrary functions that need to follow the same convention - for example, a user can add them to auxiliary functions called by the server routine that have their return value of type `kern_return_t` unconditionally returned from the routine. The attribute can be applied to C++ methods, and in this case it will be automatically applied to overrides if the method is virtual. The attribute can also be written using C++11 syntax: `[[mig::server_routine]]`.

min_vector_width

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
min_vector_width	clang::min_vector_width	clang::min_vector_width					Yes

Clang supports the `__attribute__((min_vector_width(width)))` attribute. This attribute may be attached to a function and informs the backend that this function desires vectors of at least this width to be generated. Target-specific maximum vector widths still apply. This means even if you ask for something larger than the target supports, you will only get what the target supports. This attribute is meant to be a hint to control target heuristics that may generate narrower vectors than what the target hardware supports.

This is currently used by the X86 target to allow some CPUs that support 512-bit vectors to be limited to using 256-bit vectors to avoid frequency penalties. This is currently enabled with the `-prefer-vector-width=256` command line option. The `min_vector_width` attribute can be used to prevent the backend from trying to split vector operations to match the `prefer-vector-width`. All X86 vector intrinsics from x86intrin.h already set this attribute. Additionally, use of any of the X86-specific vector builtins will implicitly set this attribute on the calling function. The intent is that explicitly writing vector code using the X86 intrinsics will prevent `prefer-vector-width` from affecting the code.

no_builtin

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
no_builtin	clang::no_builtin	clang::no_builtin					Yes

The `__attribute__((no_builtin))` is similar to the `-fno-builtin` flag except it is specific to the body of a function. The attribute may also be applied to a virtual function but has no effect on the behavior of overriding functions in a derived class.

It accepts one or more strings corresponding to the specific names of the builtins to disable (e.g. “memcpy”, “memset”). If the attribute is used without parameters it will disable all buitins at once.

```
// The compiler is not allowed to add any builtin to foo's body.
void foo(char* data, size_t count) __attribute__((no_builtin)) {
  // The compiler is not allowed to convert the loop into
  // `__builtin_memset(data, 0xFE, count);`.
  for (size_t i = 0; i < count; ++i)
    data[i] = 0xFE;
}

// The compiler is not allowed to add the `memcpy` builtin to bar's body.
void bar(char* data, size_t count) __attribute__((no_builtin("memcpy"))) {
  // The compiler is allowed to convert the loop into
  // `__builtin_memset(data, 0xFE, count);` but cannot generate any
  // `__builtin_memcpy`
  for (size_t i = 0; i < count; ++i)
    data[i] = 0xFE;
}
```

no_caller_saved_registers

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
no_caller_saved_registers	gnu::no_caller_saved_registers	gnu::no_caller_saved_registers					

Use this attribute to indicate that the specified function has no caller-saved registers. That is, all registers are callee-saved except for registers used for passing parameters to the function or returning parameters from the

function. The compiler saves and restores any modified registers that were not used for passing or returning arguments to the function.

The user can call functions specified with the 'no_caller_saved_registers' attribute from an interrupt handler without saving and restoring all call-clobbered registers.

Note that 'no_caller_saved_registers' attribute is not a calling convention. In fact, it only overrides the decision of which registers should be saved by the caller, but not how the parameters are passed from the caller to the callee.

For example:

```
__attribute__ ((no_caller_saved_registers, fastcall))
void f (int arg1, int arg2) {
  ...
}
```

In this case parameters 'arg1' and 'arg2' will be passed in registers. In this case, on 32-bit x86 targets, the function 'f' will use ECX and EDX as register parameters. However, it will not assume any scratch registers and should save and restore any modified registers except for ECX and EDX.

no_profile_instrument_function

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
no_profile_instrument_function	gnu::no_profile_instrument_function	gnu::no_profile_instrument_function					Yes

Use the `no_profile_instrument_function` attribute on a function declaration to denote that the compiler should not instrument the function with profile-related instrumentation, such as via the `-fprofile-generate` / `-fprofile-instr-generate` / `-fcs-profile-generate` / `-fprofile-arcs` flags.

no_sanitize

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
no_sanitize	clang::no_sanitize	clang::no_sanitize					Yes

Use the `no_sanitize` attribute on a function or a global variable declaration to specify that a particular instrumentation or set of instrumentations should not be applied.

The attribute takes a list of string literals with the following accepted values: `*` all values accepted by `-fno-sanitize=`; `*coverage`, to disable SanitizerCoverage instrumentation.

For example, `__attribute__((no_sanitize("address", "thread")))` specifies that AddressSanitizer and ThreadSanitizer should not be applied to the function or variable. Using `__attribute__((no_sanitize("coverage")))` specifies that SanitizerCoverage should not be applied to the function.

See Controlling Code Generation for a full list of supported sanitizer flags.

no_sanitize_address, no_address_safety_analysis

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
no_address_safety_analysis no_sanitize_address no_sanitize_thread no_sanitize_memory	gnu::no_address_safety_analysis gnu::no_sanitize_address gnu::no_sanitize_thread clang::no_sanitize_memory	gnu::no_address_safety_analysis gnu::no_sanitize_address gnu::no_sanitize_thread clang::no_sanitize_memory					Yes

Use `__attribute__((no_sanitize_address))` on a function or a global variable declaration to specify that address safety instrumentation (e.g. AddressSanitizer) should not be applied.

no_sanitize_memory

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
no_address_safety_analysis no_sanitize_address no_sanitize_thread no_sanitize_memory	gnu::no_address_safety_analysis gnu::no_sanitize_address gnu::no_sanitize_thread clang::no_sanitize_memory	gnu::no_address_safety_analysis gnu::no_sanitize_address gnu::no_sanitize_thread clang::no_sanitize_memory					Yes

Use `__attribute__((no_sanitize_memory))` on a function declaration to specify that checks for uninitialized memory should not be inserted (e.g. by MemorySanitizer). The function may still be instrumented by the tool to avoid false positives in other places.

no_sanitize_thread

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
no_address_safety_analysis no_sanitize_address no_sanitize_thread no_sanitize_memory	gnu::no_address_safety_analysis gnu::no_sanitize_address gnu::no_sanitize_thread clang::no_sanitize_memory	gnu::no_address_safety_analysis gnu::no_sanitize_address gnu::no_sanitize_thread clang::no_sanitize_memory					Yes

Use `__attribute__((no_sanitize_thread))` on a function declaration to specify that checks for data races on plain (non-atomic) memory accesses should not be inserted by ThreadSanitizer. The function is still instrumented by the tool to avoid false positives and provide meaningful stack traces.

no_speculative_load_hardening

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
no_speculative_load_hardening	clang::no_speculative_load_hardening	clang::no_speculative_load_hardening					Yes

This attribute can be applied to a function declaration in order to indicate

that Speculative Load Hardening is *not* needed for the function body. This can also be applied to a method in Objective C. This attribute will take precedence over the command line flag in the case where -mspeculative-load-hardening is specified.

Warning: This attribute may not prevent Speculative Load Hardening from being enabled for a function which inlines a function that has the 'speculative_load_hardening' attribute. This is intended to provide a maximally conservative model where the code that is marked with the 'speculative_load_hardening' attribute will always (even when inlined) be hardened. A user of this attribute may want to mark functions called by a function they do not want to be hardened with the 'noinline' attribute.

For example:

```
__attribute__((speculative_load_hardening))
int foo(int i) {
  return i;
}

// Note: bar() may still have speculative load hardening enabled if
// foo() is inlined into bar(). Mark foo() with __attribute__((noinline))
// to avoid this situation.
__attribute__((no_speculative_load_hardening))
int bar(int i) {
```

```
  return foo(i);
}
```

no_split_stack

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
no_split_stack	gnu::no_split_stack	gnu::no_split_stack					Yes

The `no_split_stack` attribute disables the emission of the split stack preamble for a particular function. It has no effect if `-fsplit-stack` is not specified.

no_stack_protector

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
no_stack_protector	clang::no_stack_protector	clang::no_stack_protector					Yes

Clang supports the `__attribute__((no_stack_protector))` attribute which disables the stack protector on the specified function. This attribute is useful for selectively disabling the stack protector on some functions when building with `-fstack-protector` compiler option.

For example, it disables the stack protector for the function `foo` but function `bar` will still be built with the stack protector with the `-fstack-protector` option.

```
int __attribute__((no_stack_protector))
foo (int x); // stack protection will be disabled for foo.

int bar(int y); // bar can be built with the stack protector.
```

noalias

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
			noalias				

The `noalias` attribute indicates that the only memory accesses inside function are loads and stores from objects pointed to by its pointer-typed arguments, with arbitrary offsets.

nocf_check

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
nocf_check	gnu::nocf_check	gnu::nocf_check					Yes

Jump Oriented Programming attacks rely on tampering with addresses used by indirect call / jmp, e.g. redirect control-flow to non-programmer intended bytes in the binary. X86 Supports Indirect Branch Tracking (IBT) as part of Control-Flow Enforcement Technology (CET). IBT instruments ENDBR instructions used to specify valid targets of indirect call / jmp. The `nocf_check` attribute has two roles: 1. Appertains to a function - do not add ENDBR instruction at the beginning of the function. 2. Appertains to a function pointer - do not track the target function of this pointer (by adding nocf_check prefix to the indirect-call instruction).

nodiscard, warn_unused_result

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
warn_unused_result	nodiscard clang::warn_unused_result gnu::warn_unused_result	nodiscard gnu::warn_unused_result					Yes

Clang supports the ability to diagnose when the results of a function call expression are discarded under suspicious circumstances. A diagnostic is generated when a function or its return type is marked with `[[nodiscard]]` (or `__attribute__((warn_unused_result))`) and the function call appears as a potentially-evaluated discarded-value expression that is not explicitly cast to `void`.

A string literal may optionally be provided to the attribute, which will be reproduced in any resulting diagnostics. Redeclarations using different forms of the attribute (with or without the string literal or with different string literal contents) are allowed. If there are redeclarations of the entity with differing string literals, it is unspecified which one will be used by Clang in any resulting diagnostics.

Additionally, discarded temporaries resulting from a call to a constructor marked with `[[nodiscard]]` or a constructor of a type marked `[[nodiscard]]` will also diagnose. This also applies to type conversions that use the annotated `[[nodiscard]]` constructor or result in an annotated type.

noduplicate

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
noduplicate	clang::noduplicate	clang::noduplicate					Yes

The `noduplicate` attribute can be placed on function declarations to control whether function calls to this function can be duplicated or not as a result of optimizations. This is required for the implementation of functions with certain

special requirements, like the OpenCL “barrier” function, that might need to be run concurrently by all the threads that are executing in lockstep on the hardware. For example this attribute applied on the function “nodupfunc” in the code below avoids that:

```
void nodupfunc() __attribute__((noduplicate));
// Setting it as a C++11 attribute is also valid
// void nodupfunc() [[clang::noduplicate]];
void foo();
void bar();

nodupfunc();
if (a > n) {
  foo();
} else {
  bar();
}
```

gets possibly modified by some optimizations into code similar to this:

```
if (a > n) {
  nodupfunc();
  foo();
} else {
  nodupfunc();
  bar();
}
```

where the call to “nodupfunc” is duplicated and sunk into the two branches of the condition.

noinline

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
noinline	gnu::noinline clang::noinline	gnu::noinline clang::noinline	noinline	__noinline__			Yes

This function attribute suppresses the inlining of a function at the call sites of the function.

`[[clang::noinline]]` spelling can be used as a statement attribute; other spellings of the attribute are not supported on statements. If a statement is marked `[[clang::noinline]]` and contains calls, those calls inside the statement will not be inlined by the compiler.

`__noinline__` can be used as a keyword in CUDA/HIP languages. This is to avoid diagnostics due to usage of `__attribute__((__noinline__))` with `__noinline__` defined as a macro as `__attribute__((noinline))`.

```
int example(void) {
  int r;
  [[clang::noinline]] foo();
  [[clang::noinline]] r = bar();
  return r;
}
```

nomicromips

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
nomicromips	gnu::nomicromips	gnu::nomicromips					Yes

Clang supports the GNU style `__attribute__((micromips))` and `__attribute__((nomicromips))` attributes on MIPS targets. These attributes may be attached to a function definition and instructs the backend to generate or not to generate microMIPS code for that function.

These attributes override the `-mmicromips` and `-mno-micromips` options on the command line.

noreturn, _Noreturn

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
	noreturn	noreturn _Noreturn					Yes

A function declared as `[[noreturn]]` shall not return to its caller. The compiler will generate a diagnostic for a function declared as `[[noreturn]]` that appears to be capable of returning to its caller.

The `[[_Noreturn]]` spelling is deprecated and only exists to ease code migration for code using `[[noreturn]]` after including `<stdnoreturn.h>`.

not_tail_called

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
not_tail_called	clang::not_tail_called	clang::not_tail_called					Yes

The `not_tail_called` attribute prevents tail-call optimization on statically bound calls. Objective-c methods, and functions marked as `always_inline` cannot be marked as `not_tail_called`.

For example, it prevents tail-call optimization in the following case:

```
int __attribute__((not_tail_called)) foo1(int);

int foo2(int a) {
  return foo1(a); // No tail-call optimization on direct calls.
}
```

However, it doesn't prevent tail-call optimization in this case:

```
int __attribute__((not_tail_called)) foo1(int);

int foo2(int a) {
  int (*fn)(int) = &foo1;
```

```
  // not_tail_called has no effect on an indirect call even if the call can
  // be resolved at compile time.
  return (*fn)(a);
}
```

Generally, marking an overriding virtual function as `not_tail_called` is not useful, because this attribute is a property of the static type. Calls made through a pointer or reference to the base class type will respect the `not_tail_called` attribute of the base class's member function, regardless of the runtime destination of the call:

```
struct Foo { virtual void f(); };
struct Bar : Foo {
  [[clang::not_tail_called]] void f() override;
};
void callera(Bar& bar) {
  Foo& foo = bar;
  // not_tail_called has no effect on here, even though the
  // underlying method is f from Bar.
  foo.f();
  bar.f(); // No tail-call optimization on here.
}
```

nothrow

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
nothrow	gnu::nothrow	gnu::nothrow	nothrow				Yes

Clang supports the GNU style `__attribute__((nothrow))` and Microsoft style `__declspec(nothrow)` attribute as an equivalent of `noexcept` on function declarations. This attribute informs the compiler that the annotated function does not throw an exception. This prevents exception-unwinding. This attribute is particularly useful on functions in the C Standard Library that are guaranteed to not throw an exception.

ns_consumed

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
ns_consumed	clang::ns_consumed	clang::ns_consumed					Yes

The behavior of a function with respect to reference counting for Foundation (Objective-C), CoreFoundation (C) and OSObject (C++) is determined by a naming convention (e.g. functions starting with "get" are assumed to return at `+0`).

It can be overridden using a family of the following attributes. In Objective-C, the annotation `__attribute__((ns_returns_retained))` applied to a function communicates that the object is returned at `+1`, and the caller is responsible for freeing it. Similarly, the annotation `__attribute__((ns_returns_not_retained))` specifies that the object is returned at `+0` and the ownership remains with the callee. The annotation `__attribute__((ns_consumes_self))` specifies that the Objective-C method call consumes the reference to `self`, e.g. by attaching it to a supplied parameter. Additionally, parameters

can have an annotation `__attribute__((ns_consumed))`, which specifies that passing an owned object as that parameter effectively transfers the ownership, and the caller is no longer responsible for it. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

In C programs using CoreFoundation, a similar set of attributes: `__attribute__((cf_returns_not_retained))`, `__attribute__((cf_returns_retained))` and `__attribute__((cf_consumed))` have the same respective semantics when applied to CoreFoundation objects. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

Finally, in C++ interacting with XNU kernel (objects inheriting from OSObject), the same attribute family is present: `__attribute__((os_returns_not_retained))`, `__attribute__((os_returns_retained))` and `__attribute__((os_consumed))`, with the same respective semantics. Similar to `__attribute__((ns_consumes_self))`, `__attribute__((os_consumes_this))` specifies that the method call consumes the reference to “this” (e.g., when attaching it to a different object supplied as a parameter). Out parameters (parameters the function is meant to write into, either via pointers-to-pointers or references-to-pointers) may be annotated with `__attribute__((os_returns_retained))` or `__attribute__((os_returns_not_retained))` which specifies that the object written into the out parameter should (or respectively should not) be released after use. Since often out parameters may or may not be written depending on the exit code of the function, annotations `__attribute__((os_returns_retained_on_zero))` and `__attribute__((os_returns_retained_on_non_zero))` specify that an out parameter at `+1` is written if and only if the function returns a zero (respectively non-zero) error code. Observe that return-code-dependent out parameter annotations are only available for retained out parameters, as non-retained object do not have to be released by the callee. These attributes are only used by the Clang Static Analyzer.

The family of attributes `X_returns_X_retained` can be added to functions, C++ methods, and Objective-C methods and properties. Attributes `X_consumed` can be added to parameters of methods, functions, and Objective-C methods.

ns_consumes_self

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
ns_consumes_self	clang::ns_consumes_self	clang::ns_consumes_self					Yes

The behavior of a function with respect to reference counting for Foundation (Objective-C), CoreFoundation (C) and OSObject (C++) is determined by a naming convention (e.g. functions starting with “get” are assumed to return at `+0`).

It can be overridden using a family of the following attributes. In Objective-C, the annotation `__attribute__((ns_returns_retained))` applied to a function communicates that the object is returned at `+1`, and the caller is responsible for freeing it. Similarly, the annotation `__attribute__((ns_returns_not_retained))` specifies that the object is returned at `+0` and the ownership remains with the callee. The annotation `__attribute__((ns_consumes_self))` specifies that the Objective-C method call consumes the reference to `self`, e.g. by attaching it to a supplied parameter. Additionally, parameters can have an annotation `__attribute__((ns_consumed))`, which specifies that passing an owned object as that parameter effectively transfers the ownership, and the caller is no longer responsible for it. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

In C programs using CoreFoundation, a similar set of attributes: `__attribute__((cf_returns_not_retained))`, `__attribute__((cf_returns_retained))` and `__attribute__((cf_consumed))` have the same respective semantics when applied to CoreFoundation objects. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

Finally, in C++ interacting with XNU kernel (objects inheriting from OSObject), the same attribute family is present: `__attribute__((os_returns_not_retained))`, `__attribute__((os_returns_retained))` and

`__attribute__((os_consumed))`, with the same respective semantics. Similar to `__attribute__((ns_consumes_self))`, `__attribute__((os_consumes_this))` specifies that the method call consumes the reference to "this" (e.g., when attaching it to a different object supplied as a parameter). Out parameters (parameters the function is meant to write into, either via pointers-to-pointers or references-to-pointers) may be annotated with `__attribute__((os_returns_retained))` or `__attribute__((os_returns_not_retained))` which specifies that the object written into the out parameter should (or respectively should not) be released after use. Since often out parameters may or may not be written depending on the exit code of the function, annotations `__attribute__((os_returns_retained_on_zero))` and `__attribute__((os_returns_retained_on_non_zero))` specify that an out parameter at `+1` is written if and only if the function returns a zero (respectively non-zero) error code. Observe that return-code-dependent out parameter annotations are only available for retained out parameters, as non-retained object do not have to be released by the callee. These attributes are only used by the Clang Static Analyzer.

The family of attributes `X_returns_X_retained` can be added to functions, C++ methods, and Objective-C methods and properties. Attributes `X_consumed` can be added to parameters of methods, functions, and Objective-C methods.

ns_returns_autoreleased

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
ns_returns_autoreleased	clang::ns_returns_autoreleased	clang::ns_returns_autoreleased					

The behavior of a function with respect to reference counting for Foundation (Objective-C), CoreFoundation (C) and OSObject (C++) is determined by a naming convention (e.g. functions starting with "get" are assumed to return at `+0`).

It can be overridden using a family of the following attributes. In Objective-C, the annotation `__attribute__((ns_returns_retained))` applied to a function communicates that the object is returned at `+1`, and the caller is responsible for freeing it. Similarly, the annotation `__attribute__((ns_returns_not_retained))` specifies that the object is returned at `+0` and the ownership remains with the callee. The annotation `__attribute__((ns_consumes_self))` specifies that the Objective-C method call consumes the reference to `self`, e.g. by attaching it to a supplied parameter. Additionally, parameters can have an annotation `__attribute__((ns_consumed))`, which specifies that passing an owned object as that parameter effectively transfers the ownership, and the caller is no longer responsible for it. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

In C programs using CoreFoundation, a similar set of attributes: `__attribute__((cf_returns_not_retained))`, `__attribute__((cf_returns_retained))` and `__attribute__((cf_consumed))` have the same respective semantics when applied to CoreFoundation objects. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

Finally, in C++ interacting with XNU kernel (objects inheriting from OSObject), the same attribute family is present: `__attribute__((os_returns_not_retained))`, `__attribute__((os_returns_retained))` and `__attribute__((os_consumed))`, with the same respective semantics. Similar to `__attribute__((ns_consumes_self))`, `__attribute__((os_consumes_this))` specifies that the method call consumes the reference to "this" (e.g., when attaching it to a different object supplied as a parameter). Out parameters (parameters the function is meant to write into, either via pointers-to-pointers or references-to-pointers) may be annotated with `__attribute__((os_returns_retained))` or `__attribute__((os_returns_not_retained))` which specifies that the object written into the out parameter should (or respectively should not) be released after use. Since often out parameters may or may not be written depending on the exit code of the function, annotations `__attribute__((os_returns_retained_on_zero))` and `__attribute__((os_returns_retained_on_non_zero))` specify that an out parameter at `+1` is written if and only if the function returns a zero (respectively non-zero) error code. Observe that return-code-dependent out

parameter annotations are only available for retained out parameters, as non-retained object do not have to be released by the callee. These attributes are only used by the Clang Static Analyzer.

The family of attributes `X_returns_X_retained` can be added to functions, C++ methods, and Objective-C methods and properties. Attributes `X_consumed` can be added to parameters of methods, functions, and Objective-C methods.

ns_returns_not_retained

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
ns_returns_not_retained	clang::ns_returns_not_retained	clang::ns_returns_not_retained					

The behavior of a function with respect to reference counting for Foundation (Objective-C), CoreFoundation (C) and OSObject (C++) is determined by a naming convention (e.g. functions starting with “get” are assumed to return at `+0`).

It can be overridden using a family of the following attributes. In Objective-C, the annotation `__attribute__((ns_returns_retained))` applied to a function communicates that the object is returned at `+1`, and the caller is responsible for freeing it. Similarly, the annotation `__attribute__((ns_returns_not_retained))` specifies that the object is returned at `+0` and the ownership remains with the callee. The annotation `__attribute__((ns_consumes_self))` specifies that the Objective-C method call consumes the reference to `self`, e.g. by attaching it to a supplied parameter. Additionally, parameters can have an annotation `__attribute__((ns_consumed))`, which specifies that passing an owned object as that parameter effectively transfers the ownership, and the caller is no longer responsible for it. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

In C programs using CoreFoundation, a similar set of attributes: `__attribute__((cf_returns_not_retained))`, `__attribute__((cf_returns_retained))` and `__attribute__((cf_consumed))` have the same respective semantics when applied to CoreFoundation objects. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

Finally, in C++ interacting with XNU kernel (objects inheriting from OSObject), the same attribute family is present: `__attribute__((os_returns_not_retained))`, `__attribute__((os_returns_retained))` and `__attribute__((os_consumed))`, with the same respective semantics. Similar to `__attribute__((ns_consumes_self))`, `__attribute__((os_consumes_this))` specifies that the method call consumes the reference to “this” (e.g., when attaching it to a different object supplied as a parameter). Out parameters (parameters the function is meant to write into, either via pointers-to-pointers or references-to-pointers) may be annotated with `__attribute__((os_returns_retained))` or `__attribute__((os_returns_not_retained))` which specifies that the object written into the out parameter should (or respectively should not) be released after use. Since often out parameters may or may not be written depending on the exit code of the function, annotations `__attribute__((os_returns_retained_on_zero))` and `__attribute__((os_returns_retained_on_non_zero))` specify that an out parameter at `+1` is written if and only if the function returns a zero (respectively non-zero) error code. Observe that return-code-dependent out parameter annotations are only available for retained out parameters, as non-retained object do not have to be released by the callee. These attributes are only used by the Clang Static Analyzer.

The family of attributes `X_returns_X_retained` can be added to functions, C++ methods, and Objective-C methods and properties. Attributes `X_consumed` can be added to parameters of methods, functions, and Objective-C methods.

ns_returns_retained

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
ns_returns_retained	clang::ns_returns_retained	clang::ns_returns_retained					

The behavior of a function with respect to reference counting for Foundation (Objective-C), CoreFoundation (C) and OSObject (C++) is determined by a naming convention (e.g. functions starting with “get” are assumed to return at +0).

It can be overridden using a family of the following attributes. In Objective-C, the annotation `__attribute__((ns_returns_retained))` applied to a function communicates that the object is returned at +1, and the caller is responsible for freeing it. Similarly, the annotation `__attribute__((ns_returns_not_retained))` specifies that the object is returned at +0 and the ownership remains with the callee. The annotation `__attribute__((ns_consumes_self))` specifies that the Objective-C method call consumes the reference to `self`, e.g. by attaching it to a supplied parameter. Additionally, parameters can have an annotation `__attribute__((ns_consumed))`, which specifies that passing an owned object as that parameter effectively transfers the ownership, and the caller is no longer responsible for it. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

In C programs using CoreFoundation, a similar set of attributes: `__attribute__((cf_returns_not_retained))`, `__attribute__((cf_returns_retained))` and `__attribute__((cf_consumed))` have the same respective semantics when applied to CoreFoundation objects. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

Finally, in C++ interacting with XNU kernel (objects inheriting from OSObject), the same attribute family is present: `__attribute__((os_returns_not_retained))`, `__attribute__((os_returns_retained))` and `__attribute__((os_consumed))`, with the same respective semantics. Similar to `__attribute__((ns_consumes_self))`, `__attribute__((os_consumes_this))` specifies that the method call consumes the reference to “this” (e.g., when attaching it to a different object supplied as a parameter). Out parameters (parameters the function is meant to write into, either via pointers-to-pointers or references-to-pointers) may be annotated with `__attribute__((os_returns_retained))` or `__attribute__((os_returns_not_retained))` which specifies that the object written into the out parameter should (or respectively should not) be released after use. Since often out parameters may or may not be written depending on the exit code of the function, annotations `__attribute__((os_returns_retained_on_zero))` and `__attribute__((os_returns_retained_on_non_zero))` specify that an out parameter at +1 is written if and only if the function returns a zero (respectively non-zero) error code. Observe that return-code-dependent out parameter annotations are only available for retained out parameters, as non-retained object do not have to be released by the callee. These attributes are only used by the Clang Static Analyzer.

The family of attributes `X_returns_X_retained` can be added to functions, C++ methods, and Objective-C methods and properties. Attributes `X_consumed` can be added to parameters of methods, functions, and Objective-C methods.

numthreads

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic

The `numthreads` attribute applies to HLSL shaders where explcit thread counts are required. The `X`, `Y`, and `Z` values provided to the attribute dictate the thread id. Total number of threads executed is `X * Y * Z`.

The full documentation is available here: https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/sm5-attributes-numthreads

objc_method_family

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
objc_method_family	clang::objc_method_family	clang::objc_method_family					Yes

Many methods in Objective-C have conventional meanings determined by their selectors. It is sometimes useful to be able to mark a method as having a particular conventional meaning despite not having the right selector, or as not having the conventional meaning that its selector would suggest. For these use cases, we provide an attribute to specifically describe the "method family" that a method belongs to.

Usage: `__attribute__((objc_method_family(X)))`, where `X` is one of `none`, `alloc`, `copy`, `init`, `mutableCopy`, or `new`. This attribute can only be placed at the end of a method declaration:

```
- (NSString *)initMyStringValue __attribute__((objc_method_family(none)));
```

Users who do not wish to change the conventional meaning of a method, and who merely want to document its non-standard retain and release semantics, should use the retaining behavior attributes (`ns_returns_retained`, `ns_returns_not_retained`, etc).

Query for this feature with `__has_attribute(objc_method_family)`.

objc_requires_super

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
objc_requires_super	clang::objc_requires_super	clang::objc_requires_super					Yes

Some Objective-C classes allow a subclass to override a particular method in a parent class but expect that the overriding method also calls the overridden method in the parent class. For these cases, we provide an attribute to designate that a method requires a "call to `super`" in the overriding method in the subclass.

Usage: `__attribute__((objc_requires_super))`. This attribute can only be placed at the end of a method declaration:

```
- (void)foo __attribute__((objc_requires_super));
```

This attribute can only be applied the method declarations within a class, and not a protocol. Currently this attribute does not enforce any placement of where the call occurs in the overriding method (such as in the case of `-dealloc` where the call must appear at the end). It checks only that it exists.

Note that on both OS X and iOS that the Foundation framework provides a convenience macro `NS_REQUIRES_SUPER` that provides syntactic sugar for this attribute:

```
- (void)foo NS_REQUIRES_SUPER;
```

This macro is conditionally defined depending on the compiler's support for this attribute. If the compiler does not support the attribute the macro expands to nothing.

Operationally, when a method has this annotation the compiler will warn if the implementation of an override in a subclass does not call super. For example:

```
warning: method possibly missing a [super AnnotMeth] call
- (void) AnnotMeth{};
                  ^
```

optnone

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
optnone	clang::optnone	clang::optnone					Yes

The `optnone` attribute suppresses essentially all optimizations on a function or method, regardless of the optimization level applied to the compilation unit as a whole. This is particularly useful when you need to debug a particular function, but it is infeasible to build the entire application without optimization. Avoiding optimization on the specified function can improve the quality of the debugging information for that function.

This attribute is incompatible with the `always_inline` and `minsize` attributes.

os_consumed

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
os_consumed	clang::os_consumed	clang::os_consumed					Yes

The behavior of a function with respect to reference counting for Foundation (Objective-C), CoreFoundation (C) and OSObject (C++) is determined by a naming convention (e.g. functions starting with “get” are assumed to return at `+0`).

It can be overridden using a family of the following attributes. In Objective-C, the annotation `__attribute__((ns_returns_retained))` applied to a function communicates that the object is returned at `+1`, and the caller is responsible for freeing it. Similarly, the annotation `__attribute__((ns_returns_not_retained))` specifies that the object is returned at `+0` and the ownership remains with the callee. The annotation `__attribute__((ns_consumes_self))` specifies that the Objective-C method call consumes the reference to `self`, e.g. by attaching it to a supplied parameter. Additionally, parameters can have an annotation `__attribute__((ns_consumed))`, which specifies that passing an owned object as that parameter effectively transfers the ownership, and the caller is no longer responsible for it. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

In C programs using CoreFoundation, a similar set of attributes: `__attribute__((cf_returns_not_retained))`, `__attribute__((cf_returns_retained))` and `__attribute__((cf_consumed))` have the same respective semantics when applied to CoreFoundation objects. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

Finally, in C++ interacting with XNU kernel (objects inheriting from OSObject), the same attribute family is present: `__attribute__((os_returns_not_retained))`, `__attribute__((os_returns_retained))` and `__attribute__((os_consumed))`, with the same respective semantics. Similar to `__attribute__((ns_consumes_self))`, `__attribute__((os_consumes_this))` specifies that the method call consumes the reference to “this” (e.g., when attaching it to a different object supplied as a parameter). Out parameters (parameters the function is meant to write into, either via pointers-to-pointers or references-to-pointers) may be annotated with `__attribute__((os_returns_retained))` or

`__attribute__((os_returns_not_retained))` which specifies that the object written into the out parameter should (or respectively should not) be released after use. Since often out parameters may or may not be written depending on the exit code of the function, annotations `__attribute__((os_returns_retained_on_zero))` and `__attribute__((os_returns_retained_on_non_zero))` specify that an out parameter at `+1` is written if and only if the function returns a zero (respectively non-zero) error code. Observe that return-code-dependent out parameter annotations are only available for retained out parameters, as non-retained object do not have to be released by the callee. These attributes are only used by the Clang Static Analyzer.

The family of attributes `X_returns_X_retained` can be added to functions, C++ methods, and Objective-C methods and properties. Attributes `X_consumed` can be added to parameters of methods, functions, and Objective-C methods.

os_consumes_this

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
os_consumes_this	clang::os_consumes_this	clang::os_consumes_this					

The behavior of a function with respect to reference counting for Foundation (Objective-C), CoreFoundation (C) and OSObject (C++) is determined by a naming convention (e.g. functions starting with “get” are assumed to return at `+0`).

It can be overridden using a family of the following attributes. In Objective-C, the annotation `__attribute__((ns_returns_retained))` applied to a function communicates that the object is returned at `+1`, and the caller is responsible for freeing it. Similarly, the annotation `__attribute__((ns_returns_not_retained))` specifies that the object is returned at `+0` and the ownership remains with the callee. The annotation `__attribute__((ns_consumes_self))` specifies that the Objective-C method call consumes the reference to `self`, e.g. by attaching it to a supplied parameter. Additionally, parameters can have an annotation `__attribute__((ns_consumed))`, which specifies that passing an owned object as that parameter effectively transfers the ownership, and the caller is no longer responsible for it. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

In C programs using CoreFoundation, a similar set of attributes: `__attribute__((cf_returns_not_retained))`, `__attribute__((cf_returns_retained))` and `__attribute__((cf_consumed))` have the same respective semantics when applied to CoreFoundation objects. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

Finally, in C++ interacting with XNU kernel (objects inheriting from OSObject), the same attribute family is present: `__attribute__((os_returns_not_retained))`, `__attribute__((os_returns_retained))` and `__attribute__((os_consumed))`, with the same respective semantics. Similar to `__attribute__((ns_consumes_self))`, `__attribute__((os_consumes_this))` specifies that the method call consumes the reference to “this” (e.g., when attaching it to a different object supplied as a parameter). Out parameters (parameters the function is meant to write into, either via pointers-to-pointers or references-to-pointers) may be annotated with `__attribute__((os_returns_retained))` or `__attribute__((os_returns_not_retained))` which specifies that the object written into the out parameter should (or respectively should not) be released after use. Since often out parameters may or may not be written depending on the exit code of the function, annotations `__attribute__((os_returns_retained_on_zero))` and `__attribute__((os_returns_retained_on_non_zero))` specify that an out parameter at `+1` is written if and only if the function returns a zero (respectively non-zero) error code. Observe that return-code-dependent out parameter annotations are only available for retained out parameters, as non-retained object do not have to be released by the callee. These attributes are only used by the Clang Static Analyzer.

The family of attributes `X_returns_X_retained` can be added to functions, C++ methods, and Objective-C methods and properties. Attributes `X_consumed` can be added to parameters of methods, functions, and Objective-C methods.

os_returns_not_retained

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
os_returns_not_retained	clang::os_returns_not_retained	clang::os_returns_not_retained					Yes

The behavior of a function with respect to reference counting for Foundation (Objective-C), CoreFoundation (C) and OSObject (C++) is determined by a naming convention (e.g. functions starting with “get” are assumed to return at `+0`).

It can be overridden using a family of the following attributes. In Objective-C, the annotation `__attribute__((ns_returns_retained))` applied to a function communicates that the object is returned at `+1`, and the caller is responsible for freeing it. Similarly, the annotation `__attribute__((ns_returns_not_retained))` specifies that the object is returned at `+0` and the ownership remains with the callee. The annotation `__attribute__((ns_consumes_self))` specifies that the Objective-C method call consumes the reference to `self`, e.g. by attaching it to a supplied parameter. Additionally, parameters can have an annotation `__attribute__((ns_consumed))`, which specifies that passing an owned object as that parameter effectively transfers the ownership, and the caller is no longer responsible for it. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

In C programs using CoreFoundation, a similar set of attributes: `__attribute__((cf_returns_not_retained))`, `__attribute__((cf_returns_retained))` and `__attribute__((cf_consumed))` have the same respective semantics when applied to CoreFoundation objects. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

Finally, in C++ interacting with XNU kernel (objects inheriting from OSObject), the same attribute family is present: `__attribute__((os_returns_not_retained))`, `__attribute__((os_returns_retained))` and `__attribute__((os_consumed))`, with the same respective semantics. Similar to `__attribute__((ns_consumes_self))`, `__attribute__((os_consumes_this))` specifies that the method call consumes the reference to “this” (e.g., when attaching it to a different object supplied as a parameter). Out parameters (parameters the function is meant to write into, either via pointers-to-pointers or references-to-pointers) may be annotated with `__attribute__((os_returns_retained))` or `__attribute__((os_returns_not_retained))` which specifies that the object written into the out parameter should (or respectively should not) be released after use. Since often out parameters may or may not be written depending on the exit code of the function, annotations `__attribute__((os_returns_retained_on_zero))` and `__attribute__((os_returns_retained_on_non_zero))` specify that an out parameter at `+1` is written if and only if the function returns a zero (respectively non-zero) error code. Observe that return-code-dependent out parameter annotations are only available for retained out parameters, as non-retained object do not have to be released by the callee. These attributes are only used by the Clang Static Analyzer.

The family of attributes `X_returns_X_retained` can be added to functions, C++ methods, and Objective-C methods and properties. Attributes `X_consumed` can be added to parameters of methods, functions, and Objective-C methods.

os_returns_retained

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
os_returns_retained	clang::os_returns_retained	clang::os_returns_retained					Yes

The behavior of a function with respect to reference counting for Foundation (Objective-C), CoreFoundation (C) and OSObject (C++) is determined by a naming convention (e.g. functions starting with “get” are assumed to return at +0).

It can be overridden using a family of the following attributes. In Objective-C, the annotation `__attribute__((ns_returns_retained))` applied to a function communicates that the object is returned at +1, and the caller is responsible for freeing it. Similarly, the annotation `__attribute__((ns_returns_not_retained))` specifies that the object is returned at +0 and the ownership remains with the callee. The annotation `__attribute__((ns_consumes_self))` specifies that the Objective-C method call consumes the reference to `self`, e.g. by attaching it to a supplied parameter. Additionally, parameters can have an annotation `__attribute__((ns_consumed))`, which specifies that passing an owned object as that parameter effectively transfers the ownership, and the caller is no longer responsible for it. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

In C programs using CoreFoundation, a similar set of attributes: `__attribute__((cf_returns_not_retained))`, `__attribute__((cf_returns_retained))` and `__attribute__((cf_consumed))` have the same respective semantics when applied to CoreFoundation objects. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

Finally, in C++ interacting with XNU kernel (objects inheriting from OSObject), the same attribute family is present: `__attribute__((os_returns_not_retained))`, `__attribute__((os_returns_retained))` and `__attribute__((os_consumed))`, with the same respective semantics. Similar to `__attribute__((ns_consumes_self))`, `__attribute__((os_consumes_this))` specifies that the method call consumes the reference to “this” (e.g., when attaching it to a different object supplied as a parameter). Out parameters (parameters the function is meant to write into, either via pointers-to-pointers or references-to-pointers) may be annotated with `__attribute__((os_returns_retained))` or `__attribute__((os_returns_not_retained))` which specifies that the object written into the out parameter should (or respectively should not) be released after use. Since often out parameters may or may not be written depending on the exit code of the function, annotations `__attribute__((os_returns_retained_on_zero))` and `__attribute__((os_returns_retained_on_non_zero))` specify that an out parameter at +1 is written if and only if the function returns a zero (respectively non-zero) error code. Observe that return-code-dependent out parameter annotations are only available for retained out parameters, as non-retained object do not have to be released by the callee. These attributes are only used by the Clang Static Analyzer.

The family of attributes `X_returns_X_retained` can be added to functions, C++ methods, and Objective-C methods and properties. Attributes `X_consumed` can be added to parameters of methods, functions, and Objective-C methods.

os_returns_retained_on_non_zero

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
os_returns_retained_on_non_zero	clang::os_returns_retained_on_non_zero	clang::os_returns_retained_on_non_zero					Yes

The behavior of a function with respect to reference counting for Foundation (Objective-C), CoreFoundation (C) and OSObject (C++) is determined by a naming convention (e.g. functions starting with “get” are assumed to return at +0).

It can be overridden using a family of the following attributes. In Objective-C, the annotation `__attribute__((ns_returns_retained))` applied to a function communicates that the object is returned at +1, and the caller is responsible for freeing it. Similarly, the annotation `__attribute__((ns_returns_not_retained))` specifies that the object is returned at +0 and the ownership remains with the callee. The annotation `__attribute__((ns_consumes_self))` specifies that the Objective-C method call consumes the reference to `self`, e.g. by attaching it to a supplied parameter. Additionally, parameters can have an annotation `__attribute__((ns_consumed))`, which specifies that passing an owned object as that parameter effectively transfers the ownership, and the caller is no longer responsible for it. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

In C programs using CoreFoundation, a similar set of attributes: `__attribute__((cf_returns_not_retained))`, `__attribute__((cf_returns_retained))` and `__attribute__((cf_consumed))` have the same respective semantics when applied to CoreFoundation objects. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

Finally, in C++ interacting with XNU kernel (objects inheriting from OSObject), the same attribute family is present: `__attribute__((os_returns_not_retained))`, `__attribute__((os_returns_retained))` and `__attribute__((os_consumed))`, with the same respective semantics. Similar to `__attribute__((ns_consumes_self))`, `__attribute__((os_consumes_this))` specifies that the method call consumes the reference to “this” (e.g., when attaching it to a different object supplied as a parameter). Out parameters (parameters the function is meant to write into, either via pointers-to-pointers or references-to-pointers) may be annotated with `__attribute__((os_returns_retained))` or `__attribute__((os_returns_not_retained))` which specifies that the object written into the out parameter should (or respectively should not) be released after use. Since often out parameters may or may not be written depending on the exit code of the function, annotations `__attribute__((os_returns_retained_on_zero))` and `__attribute__((os_returns_retained_on_non_zero))` specify that an out parameter at +1 is written if and only if the function returns a zero (respectively non-zero) error code. Observe that return-code-dependent out parameter annotations are only available for retained out parameters, as non-retained object do not have to be released by the callee. These attributes are only used by the Clang Static Analyzer.

The family of attributes `X_returns_X_retained` can be added to functions, C++ methods, and Objective-C methods and properties. Attributes `X_consumed` can be added to parameters of methods, functions, and Objective-C methods.

os_returns_retained_on_zero

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
os_returns_retained_on_zero	clang::os_returns_retained_on_zero	clang::os_returns_retained_on_zero					Yes

The behavior of a function with respect to reference counting for Foundation (Objective-C), CoreFoundation (C) and OSObject (C++) is determined by a naming convention (e.g. functions starting with “get” are assumed to return at +0).

It can be overridden using a family of the following attributes. In Objective-C, the annotation `__attribute__((ns_returns_retained))` applied to a function communicates that the object is returned at +1, and the caller is responsible for freeing it. Similarly, the annotation `__attribute__((ns_returns_not_retained))` specifies that the object is returned at +0 and the ownership remains with the callee. The annotation `__attribute__((ns_consumes_self))` specifies that the Objective-C

method call consumes the reference to `self`, e.g. by attaching it to a supplied parameter. Additionally, parameters can have an annotation `__attribute__((ns_consumed))`, which specifies that passing an owned object as that parameter effectively transfers the ownership, and the caller is no longer responsible for it. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

In C programs using CoreFoundation, a similar set of attributes: `__attribute__((cf_returns_not_retained))`, `__attribute__((cf_returns_retained))` and `__attribute__((cf_consumed))` have the same respective semantics when applied to CoreFoundation objects. These attributes affect code generation when interacting with ARC code, and they are used by the Clang Static Analyzer.

Finally, in C++ interacting with XNU kernel (objects inheriting from OSObject), the same attribute family is present: `__attribute__((os_returns_not_retained))`, `__attribute__((os_returns_retained))` and `__attribute__((os_consumed))`, with the same respective semantics. Similar to `__attribute__((ns_consumes_self))`, `__attribute__((os_consumes_this))` specifies that the method call consumes the reference to "this" (e.g., when attaching it to a different object supplied as a parameter). Out parameters (parameters the function is meant to write into, either via pointers-to-pointers or references-to-pointers) may be annotated with `__attribute__((os_returns_retained))` or `__attribute__((os_returns_not_retained))` which specifies that the object written into the out parameter should (or respectively should not) be released after use. Since often out parameters may or may not be written depending on the exit code of the function, annotations `__attribute__((os_returns_retained_on_zero))` and `__attribute__((os_returns_retained_on_non_zero))` specify that an out parameter at `+1` is written if and only if the function returns a zero (respectively non-zero) error code. Observe that return-code-dependent out parameter annotations are only available for retained out parameters, as non-retained object do not have to be released by the callee. These attributes are only used by the Clang Static Analyzer.

The family of attributes `X_returns_X_retained` can be added to functions, C++ methods, and Objective-C methods and properties. Attributes `X_consumed` can be added to parameters of methods, functions, and Objective-C methods.

overloadable

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
overloadable	clang::overloadable	clang::overloadable					Yes

Clang provides support for C++ function overloading in C. Function overloading in C is introduced using the `overloadable` attribute. For example, one might provide several overloaded versions of a `tgsin` function that invokes the appropriate standard function computing the sine of a value with `float`, `double`, or `long double` precision:

```
#include <math.h>
float __attribute__((overloadable)) tgsin(float x) { return sinf(x); }
double __attribute__((overloadable)) tgsin(double x) { return sin(x); }
long double __attribute__((overloadable)) tgsin(long double x) { return sinl(x); }
```

Given these declarations, one can call `tgsin` with a `float` value to receive a `float` result, with a `double` to receive a `double` result, etc. Function overloading in C follows the rules of C++ function overloading to pick the best overload given the call arguments, with a few C-specific semantics:

- Conversion from `float` or `double` to `long double` is ranked as a floating-point promotion (per C99) rather than as a floating-point conversion (as in C++).
- A conversion from a pointer of type `T*` to a pointer of type `U*` is considered a pointer conversion (with conversion rank) if `T` and `U` are compatible types.
- A conversion from type `T` to a value of type `U` is permitted if `T` and `U` are compatible types. This conversion is given "conversion" rank.

- If no viable candidates are otherwise available, we allow a conversion from a pointer of type `T*` to a pointer of type `U*`, where `T` and `U` are incompatible. This conversion is ranked below all other types of conversions. Please note: `U` lacking qualifiers that are present on `T` is sufficient for `T` and `U` to be incompatible.

The declaration of `overloadable` functions is restricted to function declarations and definitions. If a function is marked with the `overloadable` attribute, then all declarations and definitions of functions with that name, except for at most one (see the note below about unmarked overloads), must have the `overloadable` attribute. In addition, redeclarations of a function with the `overloadable` attribute must have the `overloadable` attribute, and redeclarations of a function without the `overloadable` attribute must *not* have the `overloadable` attribute. e.g.,

```
int f(int) __attribute__((overloadable));
float f(float); // error: declaration of "f" must have the "overloadable" attribute
int f(int); // error: redeclaration of "f" must have the "overloadable" attribute

int g(int) __attribute__((overloadable));
int g(int) { } // error: redeclaration of "g" must also have the "overloadable" attribute

int h(int);
int h(int) __attribute__((overloadable)); // error: declaration of "h" must not
                                          // have the "overloadable" attribute
```

Functions marked `overloadable` must have prototypes. Therefore, the following code is ill-formed:

```
int h() __attribute__((overloadable)); // error: h does not have a prototype
```

However, `overloadable` functions are allowed to use a ellipsis even if there are no named parameters (as is permitted in C++). This feature is particularly useful when combined with the `unavailable` attribute:

```
void honeypot(...) __attribute__((overloadable, unavailable)); // calling me is an error
```

Functions declared with the `overloadable` attribute have their names mangled according to the same rules as C++ function names. For example, the three `tgsin` functions in our motivating example get the mangled names `_Z5tgsinf`, `_Z5tgsind`, and `_Z5tgsine`, respectively. There are two caveats to this use of name mangling:

- Future versions of Clang may change the name mangling of functions overloaded in C, so you should not depend on an specific mangling. To be completely safe, we strongly urge the use of `static inline` with `overloadable` functions.
- The `overloadable` attribute has almost no meaning when used in C++, because names will already be mangled and functions are already overloadable. However, when an `overloadable` function occurs within an `extern "C"` linkage specification, it's name *will* be mangled in the same way as it would in C.

For the purpose of backwards compatibility, at most one function with the same name as other `overloadable` functions may omit the `overloadable` attribute. In this case, the function without the `overloadable` attribute will not have its name mangled.

For example:

```
// Notes with mangled names assume Itanium mangling.
int f(int);
int f(double) __attribute__((overloadable));
void foo() {
  f(5); // Emits a call to f (not _Z1fi, as it would with an overload that
        // was marked with overloadable).
  f(1.0); // Emits a call to _Z1fd.
}
```

Support for unmarked overloads is not present in some versions of clang. You may query for it using `__has_extension(overloadable_unmarked)`.

Query for this attribute with `__has_attribute(overloadable)`.

patchable_function_entry

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
patchable_function_entry	gnu::patchable_function_entry	gnu::patchable_function_entry					Yes

`__attribute__((patchable_function_entry(N,M)))` is used to generate M NOPs before the function entry and N-M NOPs after the function entry. This attribute takes precedence over the command line option `-fpatchable-function-entry=N,M`. `M` defaults to 0 if omitted.

This attribute is only supported on aarch64/aarch64-be/riscv32/riscv64/i386/x86-64 targets.

preserve_access_index

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
preserve_access_index	clang::preserve_access_index	clang::preserve_access_index					Yes

Clang supports the `__attribute__((preserve_access_index))` attribute for the BPF target. This attribute may be attached to a struct or union declaration, where if -g is specified, it enables preserving struct or union member access debuginfo indices of this struct or union, similar to clang `__builtin_preserve_access_index()`.

reinitializes

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
reinitializes	clang::reinitializes						

The `reinitializes` attribute can be applied to a non-static, non-const C++ member function to indicate that this member function reinitializes the entire object to a known state, independent of the previous state of the object.

This attribute can be interpreted by static analyzers that warn about uses of an object that has been left in an indeterminate state by a move operation. If a member function marked with the `reinitializes` attribute is called on a moved-from object, the analyzer can conclude that the object is no longer in an indeterminate state.

A typical example where this attribute would be used is on functions that clear a container class:

```
template <class T>
class Container {
public:
  ...
  [[clang::reinitializes]] void Clear();
  ...
};
```

release_capability, release_shared_capability

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
release_capability release_shared_capability release_generic_capability unlock_function	clang::release_capability clang::release_shared_capability clang::release_generic_capability clang::unlock_function						

Marks a function as releasing a capability.

retain

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
retain	gnu::retain	gnu::retain					

This attribute, when attached to a function or variable definition, prevents section garbage collection in the linker. It does not prevent other discard mechanisms, such as archive member selection, and COMDAT group resolution.

If the compiler does not emit the definition, e.g. because it was not used in the translation unit or the compiler was able to eliminate all of the uses, this attribute has no effect. This attribute is typically combined with the `used` attribute to force the definition to be emitted and preserved into the final linked image.

This attribute is only necessary on ELF targets; other targets prevent section garbage collection by the linker when using the `used` attribute alone. Using the attributes together should result in consistent behavior across targets.

This attribute requires the linker to support the `SHF_GNU_RETAIN` extension. This support is available in GNU `ld` and `gold` as of binutils 2.36, as well as in `ld.lld` 13.

shader

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic

The `shader` type attribute applies to HLSL shader entry functions to identify the shader type for the entry function. The syntax is:

```
``[shader(string-literal)]``
```

where the string literal is one of: “pixel”, “vertex”, “geometry”, “hull”, “domain”, “compute”, “raygeneration”, “intersection”, “anyhit”, “closesthit”, “miss”, “callable”, “mesh”, “amplification”. Normally the shader type is set by shader target with the `-T` option like `-Tps_6_1`. When compiling to a library target like `lib_6_3`, the shader type attribute can help the compiler to identify the shader type. It is mostly used by Raytracing shaders where shaders must be compiled into a library and linked at runtime.

short_call, near

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
short_call near	gnu::short_call gnu::near	gnu::short_call gnu::near					Yes

Clang supports the `__attribute__((long_call))`, `__attribute__((far))`, `__attribute__((short__call))`, and `__attribute__((near))` attributes on MIPS targets. These attributes may only be added to function declarations and change the code generated by the compiler when directly calling the function. The `short_call` and `near` attributes are synonyms and allow calls to the function to be made using the `jal` instruction, which requires the function to be located in the same naturally aligned 256MB segment as the caller. The `long_call` and `far` attributes are synonyms and require the use of a different call sequence that works regardless of the distance between the functions.

These attributes have no effect for position-independent code.

These attributes take priority over command line switches such as `-mlong-calls` and `-mno-long-calls`.

signal

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
signal	gnu::signal	gnu::signal					Yes

Clang supports the GNU style `__attribute__((signal))` attribute on AVR targets. This attribute may be attached to a function definition and instructs the backend to generate appropriate function entry/exit code so that it can be used directly as an interrupt service routine.

Interrupt handler functions defined with the signal attribute do not re-enable interrupts.

speculative_load_hardening

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
speculative_load_hardening	clang::speculative_load_hardening	clang::speculative_load_hardening					Yes

This attribute can be applied to a function declaration in order to indicate

that Speculative Load Hardening should be enabled for the function body. This can also be applied to a method in Objective C. This attribute will take precedence over the command line flag in the case where -mno-speculative-load-hardening is specified.

Speculative Load Hardening is a best-effort mitigation against information leak attacks that make use of control flow miss-speculation - specifically miss-speculation of whether a branch is taken or not. Typically vulnerabilities enabling such attacks are classified as “Spectre variant #1”. Notably, this does not attempt to mitigate against miss-speculation of branch target, classified as “Spectre variant #2” vulnerabilities.

When inlining, the attribute is sticky. Inlining a function that carries this attribute will cause the caller to gain the attribute. This is intended to provide a maximally conservative model where the code in a function annotated with this attribute will always (even after inlining) end up hardened.

sycl_kernel

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
sycl_kernel	clang::sycl_kernel	clang::sycl_kernel					

The `sycl_kernel` attribute specifies that a function template will be used to outline device code and to generate an OpenCL kernel. Here is a code example of the SYCL program, which demonstrates the compiler’s outlining job:

```
int foo(int x) { return ++x; }

using namespace cl::sycl;
queue Q;
buffer<int, 1> a(range<1>{1024});
Q.submit([&](handler& cgh) {
  auto A = a.get_access<access::mode::write>(cgh);
  cgh.parallel_for<init_a>(range<1>{1024}, [=](id<1> index) {
    A[index] = index[0] + foo(42);
  });
}
```

A C++ function object passed to the `parallel_for` is called a “SYCL kernel”. A SYCL kernel defines the entry point to the “device part” of the code. The compiler will emit all symbols accessible from a “kernel”. In this code example, the compiler will emit “foo” function. More details about the compilation of functions for the device part can be found in the SYCL 1.2.1 specification Section 6.4. To show to the compiler entry point to the “device part” of the code, the SYCL runtime can use the `sycl_kernel` attribute in the following way:

```
namespace cl {
namespace sycl {
class handler {
  template <typename KernelName, typename KernelType/*, ...*/>
  __attribute__((sycl_kernel)) void sycl_kernel_function(KernelType KernelFuncObj) {
    // ...
```

```
    KernelFuncObj();
  }

  template <typename KernelName, typename KernelType, int Dims>
  void parallel_for(range<Dims> NumWorkItems, KernelType KernelFunc) {
#ifdef __SYCL_DEVICE_ONLY__
    sycl_kernel_function<KernelName, KernelType, Dims>(KernelFunc);
#else
    // Host implementation
#endif
  }
};
} // namespace sycl
} // namespace cl
```

The compiler will also generate an OpenCL kernel using the function marked with the `sycl_kernel` attribute. Here is the list of SYCL device compiler expectations with regard to the function marked with the `sycl_kernel` attribute:

- The function must be a template with at least two type template parameters. The compiler generates an OpenCL kernel and uses the first template parameter as a unique name for the generated OpenCL kernel. The host application uses this unique name to invoke the OpenCL kernel generated for the SYCL kernel specialized by this name and second template parameter `KernelType` (which might be an unnamed function object type).
- The function must have at least one parameter. The first parameter is required to be a function object type (named or unnamed i.e. lambda). The compiler uses function object type fields to generate OpenCL kernel parameters.
- The function must return void. The compiler reuses the body of marked functions to generate the OpenCL kernel body, and the OpenCL kernel must return `void`.

The SYCL kernel in the previous code sample meets these expectations.

target

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
target	gnu::target	gnu::target					Yes

Clang supports the GNU style `__attribute__((target("OPTIONS")))` attribute. This attribute may be attached to a function definition and instructs the backend to use different code generation options than were passed on the command line.

The current set of options correspond to the existing “subtarget features” for the target with or without a “-mno-” in front corresponding to the absence of the feature, as well as `arch="CPU"` which will change the default “CPU” for the function.

For X86, the attribute also allows `tune="CPU"` to optimize the generated code for the given CPU without changing the available instructions.

For AArch64, the attribute also allows the “branch-protection=<args>” option, where the permissible arguments and their effect on code generation are the same as for the command-line option `-mbranch-protection`.

Example “subtarget features” from the x86 backend include: “mmx”, “sse”, “sse4.2”, “avx”, “xop” and largely correspond to the machine specific options handled by the front end.

Additionally, this attribute supports function multiversioning for ELF based x86/x86-64 targets, which can be used to create multiple implementations of the same function that will be resolved at runtime based on the priority of their `target` attribute strings. A function is considered a multiversioned function if either two declarations of the function have different `target` attribute strings, or if it has a `target` attribute string of `default`. For example:

```
__attribute__((target("arch=atom")))
void foo() {} // will be called on 'atom' processors.
__attribute__((target("default")))
void foo() {} // will be called on any other processors.
```

All multiversioned functions must contain a `default` (fallback) implementation, otherwise usages of the function are considered invalid. Additionally, a function may not become multiversioned after its first use.

target_clones

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
target_clones	gnu::target_clones	gnu::target_clones					Yes

Clang supports the `target_clones("OPTIONS")` attribute. This attribute may be attached to a function declaration and causes function multiversioning, where multiple versions of the function will be emitted with different code generation options. Additionally, these versions will be resolved at runtime based on the priority of their attribute options. All `target_clone` functions are considered multiversioned functions.

All multiversioned functions must contain a `default` (fallback) implementation, otherwise usages of the function are considered invalid. Additionally, a function may not become multiversioned after its first use.

The options to `target_clones` can either be a target-specific architecture (specified as `arch=CPU`), or one of a list of subtarget features.

Example “subtarget features” from the x86 backend include: “mmx”, “sse”, “sse4.2”, “avx”, “xop” and largely correspond to the machine specific options handled by the front end.

The versions can either be listed as a comma-separated sequence of string literals or as a single string literal containing a comma-separated list of versions. For compatibility with GCC, the two formats can be mixed. For example, the following will emit 4 versions of the function:

```
__attribute__((target_clones("arch=atom,avx2","arch=ivybridge","default")))
void foo() {}
```

try_acquire_capability, try_acquire_shared_capability

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
try_acquire_capability try_acquire_shared_capability	clang::try_acquire_capability clang::try_acquire_shared_capability						

Marks a function that attempts to acquire a capability. This function may fail to actually acquire the capability; they accept a Boolean value determining whether acquiring the capability means success (true), or failing to acquire the capability means success (false).

used

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
used	gnu::used	gnu::used					

This attribute, when attached to a function or variable definition, indicates that there may be references to the entity which are not apparent in the source code. For example, it may be referenced from inline `asm`, or it may be found through a dynamic symbol or section lookup.

The compiler must emit the definition even if it appears to be unused, and it must not apply optimizations which depend on fully understanding how the entity is used.

Whether this attribute has any effect on the linker depends on the target and the linker. Most linkers support the feature of section garbage collection (`--gc-sections`), also known as “dead stripping” (`ld64 -dead_strip`) or discarding unreferenced sections (`link.exe /OPT:REF`). On COFF and Mach-O targets (Windows and Apple platforms), the *used* attribute prevents symbols from being removed by linker section GC. On ELF targets, it has no effect on its own, and the linker may remove the definition if it is not otherwise referenced. This linker GC can be avoided by also adding the `retain` attribute. Note that `retain` requires special support from the linker; see that attribute’s documentation for further information.

xray_always_instrument, xray_never_instrument, xray_log_args

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
xray_always_instrument xray_never_instrument	clang::xray_always_instrument clang::xray_never_instrument	clang::xray_always_instrument clang::xray_never_instrument					Yes

`__attribute__((xray_always_instrument))` or `[[clang::xray_always_instrument]]` is used to mark member functions (in C++), methods (in Objective C), and free functions (in C, C++, and Objective C) to be instrumented with XRay. This will cause the function to always have space at the beginning and exit points to allow for runtime patching.

Conversely, `__attribute__((xray_never_instrument))` or `[[clang::xray_never_instrument]]` will inhibit the insertion of these instrumentation points.

If a function has neither of these attributes, they become subject to the XRay heuristics used to determine whether a function should be instrumented or otherwise.

`__attribute__((xray_log_args(N)))` or `[[clang::xray_log_args(N)]]` is used to preserve N function arguments for the logging function. Currently, only N==1 is supported.

xray_always_instrument, xray_never_instrument, xray_log_args

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
xray_log_args	clang::xray_log_args	clang::xray_log_args					Yes

`__attribute__((xray_always_instrument))` or `[[clang::xray_always_instrument]]` is used to mark member functions (in C++), methods (in Objective C), and free functions (in C, C++, and Objective C) to be instrumented with XRay. This will cause the function to always have space at the beginning and exit points to allow for runtime patching.

Conversely, `__attribute__((xray_never_instrument))` or `[[clang::xray_never_instrument]]` will inhibit the insertion of these instrumentation points.

If a function has neither of these attributes, they become subject to the XRay heuristics used to determine whether a function should be instrumented or otherwise.

`__attribute__((xray_log_args(N)))` or `[[clang::xray_log_args(N)]]` is used to preserve N function arguments for the logging function. Currently, only N==1 is supported.

zero_call_used_regs

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
zero_call_used_regs	gnu::zero_call_used_regs	gnu::zero_call_used_regs					Yes

This attribute, when attached to a function, causes the compiler to zero a subset of all call-used registers before the function returns. It's used to increase program security by either mitigating Return-Oriented Programming (ROP) attacks or preventing information leakage through registers.

The term "call-used" means registers which are not guaranteed to be preserved unchanged for the caller by the current calling convention. This could also be described as "caller-saved" or "not callee-saved".

The *choice* parameters gives the programmer flexibility to choose the subset of the call-used registers to be zeroed:

- `skip` doesn't zero any call-used registers. This choice overrides any command-line arguments.
- `used` only zeros call-used registers used in the function. By `used`, we mean a register whose contents have been set or referenced in the function.
- `used-gpr` only zeros call-used GPR registers used in the funciton.
- `used-arg` only zeros call-used registers used to pass arguments to the function.
- `used-gpr-arg` only zeros call-used GPR registers used to pass arguments to the function.
- `all` zeros all call-used registers.
- `all-gpr` zeros all call-used GPR registers.
- `all-arg` zeros all call-used registers used to pass arguments to the function.
- `all-gpr-arg` zeros all call-used GPR registers used to pass arguments to the function.

The default for the attribute is contolled by the `-fzero-call-used-regs` flag.

Handle Attributes

Handles are a way to identify resources like files, sockets, and processes. They are more opaque than pointers and widely used in system programming. They have similar risks such as never releasing a resource associated with a handle, attempting to use a handle that was already released, or trying to release a handle twice. Using the annotations below it is possible to make the ownership of the handles clear: whose responsibility is to release them. They can also aid static analysis tools to find bugs.

acquire_handle

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
acquire_handle	clang::acquire_handle	clang::acquire_handle					Yes

If this annotation is on a function or a function type it is assumed to return a new handle. In case this annotation is on an output parameter, the function is assumed to fill the corresponding argument with a new handle. The attribute requires a string literal argument which used to identify the handle with later uses of `use_handle` or `release_handle`.

```
// Output arguments from Zircon.
zx_status_t zx_socket_create(uint32_t options,
                             zx_handle_t __attribute__((acquire_handle("zircon"))) * out0,
                             zx_handle_t* out1 [[clang::acquire_handle("zircon")]]);


// Returned handle.
[[clang::acquire_handle("tag")]] int open(const char *path, int oflag, ... );
int open(const char *path, int oflag, ... ) __attribute__((acquire_handle("tag")));
```

release_handle

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
release_handle	clang::release_handle	clang::release_handle					Yes

If a function parameter is annotated with `release_handle(tag)` it is assumed to close the handle. It is also assumed to require an open handle to work with. The attribute requires a string literal argument to identify the handle being released.

```
zx_status_t zx_handle_close(zx_handle_t handle [[clang::release_handle("tag")]]);
```

use_handle

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
use_handle	clang::use_handle	clang::use_handle					Yes

A function taking a handle by value might close the handle. If a function parameter is annotated with `use_handle(tag)` it is assumed to not to change the state of the handle. It is also assumed to require an open handle to work with. The attribute requires a string literal argument to identify the handle being used.

```
zx_status_t zx_port_wait(zx_handle_t handle [[clang::use_handle("zircon")]],
                         zx_time_t deadline,
                         zx_port_packet_t* packet);
```

Nullability Attributes

Whether a particular pointer may be “null” is an important concern when working with pointers in the C family of languages. The various nullability attributes indicate whether a particular pointer can be null or not, which makes APIs more expressive and can help static analysis tools identify bugs involving null pointers. Clang supports several kinds of nullability attributes: the `nonnull` and `returns_nonnull` attributes indicate which function or method parameters and result types can never be null, while nullability type qualifiers indicate which pointer types can be null (`_Nullable`) or cannot be null (`_Nonnull`).

The nullability (type) qualifiers express whether a value of a given pointer type can be null (the `_Nullable` qualifier), doesn’t have a defined meaning for null (the `_Nonnull` qualifier), or for which the purpose of null is unclear (the `_Null_unspecified` qualifier). Because nullability qualifiers are expressed within the type system, they are more general than the `nonnull` and `returns_nonnull` attributes, allowing one to express (for example) a nullable pointer to an array of nonnull pointers. Nullability qualifiers are written to the right of the pointer to which they apply. For example:

```
// No meaningful result when 'ptr' is null (here, it happens to be undefined behavior).
int fetch(int * _Nonnull ptr) { return *ptr; }

// 'ptr' may be null.
int fetch_or_zero(int * _Nullable ptr) {
  return ptr ? *ptr : 0;
}

// A nullable pointer to non-null pointers to const characters.
const char *join_strings(const char * _Nonnull * _Nullable strings, unsigned n);
```

In Objective-C, there is an alternate spelling for the nullability qualifiers that can be used in Objective-C methods and properties using context-sensitive, non-underscored keywords. For example:

```
@interface NSView : NSResponder
  - (nullable NSView *)ancestorSharedWithView:(nonnull NSView *)aView;
  @property (assign, nullable) NSView *superview;
  @property (readonly, nonnull) NSArray *subviews;
@end
```

_Nonnull

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
				_Nonnull			

The `_Nonnull` nullability qualifier indicates that null is not a meaningful value for a value of the `_Nonnull` pointer type. For example, given a declaration such as:

```
int fetch(int * _Nonnull ptr);
```

a caller of `fetch` should not provide a null value, and the compiler will produce a warning if it sees a literal null value passed to `fetch`. Note that, unlike the declaration attribute `nonnull`, the presence of `_Nonnull` does not imply that passing null is undefined behavior: `fetch` is free to consider null undefined behavior or (perhaps for backward-compatibility reasons) defensively handle null.

_Null_unspecified

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
				_Null_unspecified			

The `_Null_unspecified` nullability qualifier indicates that neither the `_Nonnull` nor `_Nullable` qualifiers make sense for a particular pointer type. It is used primarily to indicate that the role of null with specific pointers in a nullability-annotated header is unclear, e.g., due to overly-complex implementations or historical factors with a long-lived API.

_Nullable

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
				_Nullable			

The `_Nullable` nullability qualifier indicates that a value of the `_Nullable` pointer type can be null. For example, given:

```
int fetch_or_zero(int * _Nullable ptr);
```

a caller of `fetch_or_zero` can provide null.

_Nullable_result

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
				_Nullable_result			

The `_Nullable_result` nullability qualifier means that a value of the `_Nullable_result` pointer can be `nil`, just like `_Nullable`. Where this attribute differs from `_Nullable` is when it's used on a parameter to a completion handler in a Swift async method. For instance, here:

```
-(void)fetchSomeDataWithID:(int)identifier
        completionHandler:(void (^)(Data *_Nullable_result result, NSError *error))completionHandler;
```

This method asynchronously calls `completionHandler` when the data is available, or calls it with an error. `_Nullable_result` indicates to the Swift importer that this is the uncommon case where `result` can get `nil` even if no error has occurred, and will therefore import it as a Swift optional type. Otherwise, if `result` was annotated with `_Nullable`, the Swift importer will assume that `result` will always be non-nil unless an error occurred.

nonnull

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
nonnull	gnu::nonnull	gnu::nonnull					

The `nonnull` attribute indicates that some function parameters must not be null, and can be used in several different ways. It's original usage (from GCC) is as a function (or Objective-C method) attribute that specifies which parameters of the function are nonnull in a comma-separated list. For example:

```
extern void * my_memcpy (void *dest, const void *src, size_t len)
                  __attribute__((nonnull (1, 2)));
```

Here, the `nonnull` attribute indicates that parameters 1 and 2 cannot have a null value. Omitting the parenthesized list of parameter indices means that all parameters of pointer type cannot be null:

```
extern void * my_memcpy (void *dest, const void *src, size_t len)
                  __attribute__((nonnull));
```

Clang also allows the `nonnull` attribute to be placed directly on a function (or Objective-C method) parameter, eliminating the need to specify the parameter index ahead of type. For example:

```
extern void * my_memcpy (void *dest __attribute__((nonnull)),
                         const void *src __attribute__((nonnull)), size_t len);
```

Note that the `nonnull` attribute indicates that passing null to a non-null parameter is undefined behavior, which the optimizer may take advantage of to, e.g., remove null checks. The `_Nonnull` type qualifier indicates that a pointer cannot be null in a more general manner (because it is part of the type system) and does not imply undefined behavior, making it more widely applicable.

returns_nonnull

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
returns_nonnull	gnu::returns_nonnull	gnu::returns_nonnull					Yes

The `returns_nonnull` attribute indicates that a particular function (or Objective-C method) always returns a non-null pointer. For example, a particular system `malloc` might be defined to terminate a process when memory is not available rather than returning a null pointer:

```
extern void * malloc (size_t size) __attribute__((returns_nonnull));
```

The `returns_nonnull` attribute implies that returning a null pointer is undefined behavior, which the optimizer may take advantage of. The `_Nonnull` type qualifier indicates that a pointer cannot be null in a more general manner (because it is part of the type system) and does not imply undefined behavior, making it more widely applicable

OpenCL Address Spaces

The address space qualifier may be used to specify the region of memory that is used to allocate the object. OpenCL supports the following address spaces: __generic(generic), __global(global), __local(local), __private(private), __constant(constant).

```
__constant int c = ...;

__generic int* foo(global int* g) {
  __local int* l;
  private int p;
  ...
  return l;
}
```

More details can be found in the OpenCL C language Spec v2.0, Section 6.5.

[[clang::opencl_global_device]], [[clang::opencl_global_host]]

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
opencl_global_device	clang::opencl_global_device	clang::opencl_global_device					

The `global_device` and `global_host` address space attributes specify that an object is allocated in global memory on the device/host. It helps to distinguish USM (Unified Shared Memory) pointers that access global device memory from those that access global host memory. These new address spaces are a subset of the `__global/opencl_global` address space, the full address space set model for OpenCL 2.0 with the extension looks as follows:

generic->global->host
->device
->private
->local
constant

As `global_device` and `global_host` are a subset of `__global/opencl_global` address spaces it is allowed to convert `global_device` and `global_host` address spaces to `__global/opencl_global` address spaces (following ISO/IEC TR 18037 5.1.3 “Address space nesting and rules for pointers”).

[[clang::opencl_global_device]], [[clang::opencl_global_host]]

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
opencl_global_host	clang::opencl_global_host	clang::opencl_global_host					

The global_device and global_host address space attributes specify that an object is allocated in global memory on the device/host. It helps to distinguish USM (Unified Shared Memory) pointers that access global device memory from those that access global host memory. These new address spaces are a subset of the __global/opencl_global address space, the full address space set model for OpenCL 2.0 with the extension looks as follows:

generic->global->host
->device
->private
->local
constant

As global_device and global_host are a subset of __global/opencl_global address spaces it is allowed to convert global_device and global_host address spaces to __global/opencl_global address spaces (following ISO/IEC TR 18037 5.1.3 “Address space nesting and rules for pointers”).

__constant, constant, [[clang::opencl_constant]]

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
opencl_constant	clang::opencl_constant	clang::opencl_constant		__constant constant			

The constant address space attribute signals that an object is located in a constant (non-modifiable) memory region. It is available to all work items. Any type can be annotated with the constant address space attribute. Objects with the constant address space qualifier can be declared in any scope and must have an initializer.

__generic, generic, [[clang::opencl_generic]]

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
opencl_generic	clang::opencl_generic	clang::opencl_generic		__generic generic			

The generic address space attribute is only available with OpenCL v2.0 and later. It can be used with pointer types. Variables in global and local scope and function parameters in non-kernel functions can have the generic address space type attribute. It is intended to be a placeholder for any other address space except for ‘__constant’ in OpenCL code which can be used with multiple address spaces.

__global, global, [[clang::opencl_global]]

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
opencl_global	clang::opencl_global	clang::opencl_global		__global global			

The global address space attribute specifies that an object is allocated in global memory, which is accessible by all work items. The content stored in this memory area persists between kernel executions. Pointer types to the global address space are allowed as function parameters or local variables. Starting with OpenCL v2.0, the global address space can be used with global (program scope) variables and static local variable as well.

__local, local, [[clang::opencl_local]]

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
opencl_local	clang::opencl_local	clang::opencl_local		__local local			

The local address space specifies that an object is allocated in the local (work group) memory area, which is accessible to all work items in the same work group. The content stored in this memory region is not accessible after the kernel execution ends. In a kernel function scope, any variable can be in the local address space. In other scopes, only pointer types to the local address space are allowed. Local address space variables cannot have an initializer.

__private, private, [[clang::opencl_private]]

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
opencl_private	clang::opencl_private	clang::opencl_private		__private private			

The private address space specifies that an object is allocated in the private (work item) memory. Other work items cannot access the same memory area and its content is destroyed after work item execution ends. Local variables can be declared in the private address space. Function arguments are always in the private address space. Kernel function arguments of a pointer or an array type cannot point to the private address space.

Statement Attributes

#pragma clang loop

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
					clang loop unroll nounroll unroll_and_jam unrolland fuse nounrollandfuse nosimd nounroll_and_jam		

The `#pragma clang loop` directive allows loop optimization hints to be specified for the subsequent loop. The directive allows pipelining to be disabled, or vectorization, vector predication, interleaving, and unrolling to be enabled or disabled. Vector width, vector predication, interleave count, unrolling count, and the initiation interval for pipelining can be explicitly specified. See language extensions for details.

#pragma unroll, #pragma nounroll

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
					clang loop unroll nounroll unroll_and_jam unrolland fuse nounrollandfuse nosimd nounroll_and_jam		

Loop unrolling optimization hints can be specified with `#pragma unroll` and `#pragma nounroll`. The pragma is placed immediately before a for, while, do-while, or c++11 range-based for loop. GCC's loop unrolling hints `#pragma GCC unroll` and `#pragma GCC nounroll` are also supported and have identical semantics to `#pragma unroll` and `#pragma nounroll`.

Specifying `#pragma unroll` without a parameter directs the loop unroller to attempt to fully unroll the loop if the trip count is known at compile time and attempt to partially unroll the loop if the trip count is not known at compile time:

```
#pragma unroll
for (...) {
  ...
}
```

Specifying the optional parameter, `#pragma unroll _value_`, directs the unroller to unroll the loop `_value_` times. The parameter may optionally be enclosed in parentheses:

```
#pragma unroll 16
for (...) {
  ...
}

#pragma unroll(16)
for (...) {
  ...
}
```

Specifying `#pragma nounroll` indicates that the loop should not be unrolled:

```
#pragma nounroll
for (...) {
  ...
}
```

`#pragma unroll` and `#pragma unroll _value_` have identical semantics to `#pragma clang loop unroll(full)` and `#pragma clang loop unroll_count(_value_)` respectively. `#pragma nounroll` is equivalent to `#pragma clang loop unroll(disable)`. See language extensions for further details including limitations of the unroll hints.

__read_only, __write_only, __read_write (read_only, write_only, read_write)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
				__read_only read_only __write_only write_only __read_write read_write			

The access qualifiers must be used with image object arguments or pipe arguments to declare if they are being read or written by a kernel or function.

The read_only/__read_only, write_only/__write_only and read_write/__read_write names are reserved for use as access qualifiers and shall not be used otherwise.

```
kernel void
foo (read_only image2d_t imageA,
     write_only image2d_t imageB) {
  ...
}
```

In the above example imageA is a read-only 2D image object, and imageB is a write-only 2D image object.

The read_write (or __read_write) qualifier can not be used with pipe.

More details can be found in the OpenCL C language Spec v2.0, Section 6.6.

fallthrough

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
fallthrough	fallthrough clang::fallthrough gnu::fallthrough	fallthrough gnu::fallthrough					

The `fallthrough` (or `clang::fallthrough`) attribute is used to annotate intentional fall-through between switch labels. It can only be applied to a null statement placed at a point of execution between any statement and the next switch label. It is common to mark these places with a specific comment, but this attribute is meant to replace comments with a more strict annotation, which can be checked by the compiler. This attribute doesn't change semantics of the code and can be used wherever an intended fall-through occurs. It is designed to mimic control-flow statements like `break;`, so it can be placed in most places where `break;` can, but only if there are no statements on the execution path between it and the next switch label.

By default, Clang does not warn on unannotated fallthrough from one `switch` case to another. Diagnostics on fallthrough without a corresponding annotation can be enabled with the `-Wimplicit-fallthrough` argument.

Here is an example:

```
// compile with -Wimplicit-fallthrough
switch (n) {
case 22:
case 33:  // no warning: no statements between case labels
  f();
case 44:  // warning: unannotated fall-through
  g();
  [[clang::fallthrough]];
case 55:  // no warning
  if (x) {
    h();
    break;
  }
  else {
    i();
    [[clang::fallthrough]];
  }
case 66:  // no warning
  p();
  [[clang::fallthrough]]; // warning: fallthrough annotation does not
                          //          directly precede case label
  q();
case 77:  // warning: unannotated fall-through
  r();
}
```

intel_reqd_sub_group_size

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
intel_reqd_sub_group_size							Yes

The optional attribute intel_reqd_sub_group_size can be used to indicate that the kernel must be compiled and executed with the specified subgroup size. When this attribute is present, get_max_sub_group_size() is guaranteed to return the specified integer value. This is important for the correctness of many subgroup algorithms, and in some cases may be used by the compiler to generate more optimal code. See *cl_intel_required_subgroup_size <https://www.khronos.org/registry/OpenCL/extensions/intel/cl_intel_required_subgroup_size.txt>* for details.

likely and unlikely

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
	likely	clang::likely					

The `likely` and `unlikely` attributes are used as compiler hints. The attributes are used to aid the compiler to determine which branch is likely or unlikely to be taken. This is done by marking the branch substatement with one of the two attributes.

It isn't allowed to annotate a single statement with both `likely` and `unlikely`. Annotating the `true` and `false` branch of an `if` statement with the same likelihood attribute will result in a diagnostic and the attributes are ignored on both branches.

In a `switch` statement it's allowed to annotate multiple `case` labels or the `default` label with the same likelihood attribute. This makes * all labels without an attribute have a neutral likelihood, * all labels marked `[[likely]]` have an equally positive likelihood, and * all labels marked `[[unlikely]]` have an equally negative likelihood. The neutral likelihood is the more likely of path execution than the negative likelihood. The positive likelihood is the more likely of path of execution than the neutral likelihood.

These attributes have no effect on the generated code when using PGO (Profile-Guided Optimization) or at optimization level 0.

In Clang, the attributes will be ignored if they're not placed on * the `case` or `default` label of a `switch` statement, * or on the substatement of an `if` or `else` statement, * or on the substatement of an `for` or `while` statement. The C++ Standard recommends to honor them on every statement in the path of execution, but that can be confusing:

```
if (b) {
  [[unlikely]] --b; // In the path of execution,
                    // this branch is considered unlikely.
}

if (b) {
  --b;
  if(b)
    return;
  [[unlikely]] --b; // Not in the path of execution,
}                   // the branch has no likelihood information.
```

```
if (b) {
  --b;
  foo(b);
  // Whether or not the next statement is in the path of execution depends
  // on the declaration of foo():
  // In the path of execution: void foo(int);
  // Not in the path of execution: [[noreturn]] void foo(int);
  // This means the likelihood of the branch depends on the declaration
  // of foo().
  [[unlikely]] --b;
}
```

Below are some example usages of the likelihood attributes and their effects:

```
if (b) [[likely]] { // Placement on the first statement in the branch.
  // The compiler will optimize to execute the code here.
} else {
}

if (b)
  [[unlikely]] b++; // Placement on the first statement in the branch.
else {
  // The compiler will optimize to execute the code here.
}

if (b) {
  [[unlikely]] b++; // Placement on the second statement in the branch.
}                   // The attribute will be ignored.

if (b) [[likely]] {
  [[unlikely]] b++; // No contradiction since the second attribute
}                   // is ignored.

if (b)
  ;
else [[likely]] {
  // The compiler will optimize to execute the code here.
}

if (b)
  ;
else
  // The compiler will optimize to execute the next statement.
  [[likely]] b = f();

if (b) [[likely]]; // Both branches are likely. A diagnostic is issued
else [[likely]];   // and the attributes are ignored.

if (b)
  [[likely]] int i = 5; // Issues a diagnostic since the attribute
                        // isn't allowed on a declaration.

switch (i) {
  [[likely]] case 1:    // This value is likely
    ...
    break;

  [[unlikely]] case 2:  // This value is unlikely
    ...
    [[fallthrough]];
```

```
  case 3:               // No likelihood attribute
    ...
    [[likely]] break;   // No effect

  case 4: [[likely]] {  // attribute on substatement has no effect
    ...
    break;
    }

  [[unlikely]] default: // All other values are unlikely
    ...
    break;
}

switch (i) {
  [[likely]] case 0:    // This value and code path is likely
    ...
    [[fallthrough]];

  case 1:               // No likelihood attribute, code path is neutral
    break;              // falling through has no effect on the likelihood

  case 2:               // No likelihood attribute, code path is neutral
    [[fallthrough]];

  [[unlikely]] default: // This value and code path are both unlikely
    break;
}

for(int i = 0; i != size; ++i) [[likely]] {
  ...                   // The loop is the likely path of execution
}

for(const auto &E : Elements) [[likely]] {
  ...                   // The loop is the likely path of execution
}

while(i != size) [[unlikely]] {
  ...                   // The loop is the unlikely path of execution
}                       // The generated code will optimize to skip the loop body

while(true) [[unlikely]] {
  ...                   // The attribute has no effect
}                       // Clang elides the comparison and generates an infinite
                        // loop
```

likely and unlikely

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
	unlikely	clang::unlikely					

The `likely` and `unlikely` attributes are used as compiler hints. The attributes are used to aid the compiler to determine which branch is likely or unlikely to be taken. This is done by marking the branch substatement with one of the two attributes.

It isn't allowed to annotate a single statement with both `likely` and `unlikely`. Annotating the `true` and `false` branch of an `if` statement with the same likelihood attribute will result in a diagnostic and the attributes are ignored on both branches.

In a `switch` statement it's allowed to annotate multiple `case` labels or the `default` label with the same likelihood attribute. This makes * all labels without an attribute have a neutral likelihood, * all labels marked `[[likely]]` have an equally positive likelihood, and * all labels marked `[[unlikely]]` have an equally negative likelihood. The neutral likelihood is the more likely of path execution than the negative likelihood. The positive likelihood is the more likely of path of execution than the neutral likelihood.

These attributes have no effect on the generated code when using PGO (Profile-Guided Optimization) or at optimization level 0.

In Clang, the attributes will be ignored if they're not placed on * the `case` or `default` label of a `switch` statement, * or on the substatement of an `if` or `else` statement, * or on the substatement of an `for` or `while` statement. The C++ Standard recommends to honor them on every statement in the path of execution, but that can be confusing:

```
if (b) {
  [[unlikely]] --b; // In the path of execution,
                    // this branch is considered unlikely.
}

if (b) {
  --b;
  if(b)
    return;
  [[unlikely]] --b; // Not in the path of execution,
}                   // the branch has no likelihood information.

if (b) {
  --b;
  foo(b);
  // Whether or not the next statement is in the path of execution depends
  // on the declaration of foo():
  // In the path of execution: void foo(int);
  // Not in the path of execution: [[noreturn]] void foo(int);
  // This means the likelihood of the branch depends on the declaration
  // of foo().
  [[unlikely]] --b;
}
```

Below are some example usages of the likelihood attributes and their effects:

```
if (b) [[likely]] { // Placement on the first statement in the branch.
  // The compiler will optimize to execute the code here.
} else {
}

if (b)
  [[unlikely]] b++; // Placement on the first statement in the branch.
else {
  // The compiler will optimize to execute the code here.
}

if (b) {
  [[unlikely]] b++; // Placement on the second statement in the branch.
}                   // The attribute will be ignored.

if (b) [[likely]] {
  [[unlikely]] b++; // No contradiction since the second attribute
```

```
}                   // is ignored.

if (b)
  ;
else [[likely]] {
  // The compiler will optimize to execute the code here.
}

if (b)
  ;
else
  // The compiler will optimize to execute the next statement.
  [[likely]] b = f();

if (b) [[likely]]; // Both branches are likely. A diagnostic is issued
else [[likely]];   // and the attributes are ignored.

if (b)
  [[likely]] int i = 5; // Issues a diagnostic since the attribute
                        // isn't allowed on a declaration.

switch (i) {
  [[likely]] case 1:    // This value is likely
    ...
    break;

  [[unlikely]] case 2:  // This value is unlikely
    ...
    [[fallthrough]];

  case 3:               // No likelihood attribute
    ...
    [[likely]] break;   // No effect

  case 4: [[likely]] {  // attribute on substatement has no effect
    ...
    break;
    }

  [[unlikely]] default: // All other values are unlikely
    ...
    break;
}

switch (i) {
  [[likely]] case 0:    // This value and code path is likely
    ...
    [[fallthrough]];

  case 1:               // No likelihood attribute, code path is neutral
    break;              // falling through has no effect on the likelihood

  case 2:               // No likelihood attribute, code path is neutral
    [[fallthrough]];

  [[unlikely]] default: // This value and code path are both unlikely
    break;
}

for(int i = 0; i != size; ++i) [[likely]] {
```

```
  ...               // The loop is the likely path of execution
}

for(const auto &E : Elements) [[likely]] {
  ...               // The loop is the likely path of execution
}

while(i != size) [[unlikely]] {
  ...               // The loop is the unlikely path of execution
}                   // The generated code will optimize to skip the loop body

while(true) [[unlikely]] {
  ...               // The attribute has no effect
}                   // Clang elides the comparison and generates an infinite
                    // loop
```

musttail

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
musttail	clang::musttail	clang::musttail					

If a `return` statement is marked `musttail`, this indicates that the compiler must generate a tail call for the program to be correct, even when optimizations are disabled. This guarantees that the call will not cause unbounded stack growth if it is part of a recursive cycle in the call graph.

If the callee is a virtual function that is implemented by a thunk, there is no guarantee in general that the thunk tail-calls the implementation of the virtual function, so such a call in a recursive cycle can still result in unbounded stack growth.

`clang::musttail` can only be applied to a `return` statement whose value is the result of a function call (even functions returning void must use `return`, although no value is returned). The target function must have the same number of arguments as the caller. The types of the return value and all arguments must be similar according to C++ rules (differing only in cv qualifiers or array size), including the implicit “this” argument, if any. Any variables in scope, including all arguments to the function and the return value must be trivially destructible. The calling convention of the caller and callee must match, and they must not be variadic functions or have old style K&R C function declarations.

nomerge

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
nomerge	clang::nomerge	clang::nomerge					Yes

If a statement is marked `nomerge` and contains call expressions, those call expressions inside the statement will not be merged during optimization. This attribute can be used to prevent the optimizer from obscuring the source location of certain calls. For example, it will prevent tail merging otherwise identical code sequences that raise an exception or terminate the program. Tail merging normally reduces the precision of source location information, making stack traces less useful for debugging. This attribute gives the user control over the tradeoff between code size and debug information precision.

nomerge attribute can also be used as function attribute to prevent all calls to the specified function from merging. It has no effect on indirect calls.

opencl_unroll_hint

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
opencl_unroll_hint							

The opencl_unroll_hint attribute qualifier can be used to specify that a loop (for, while and do loops) can be unrolled. This attribute qualifier can be used to specify full unrolling or partial unrolling by a specified amount. This is a compiler hint and the compiler may ignore this directive. See OpenCL v2.0 s6.11.5 for details.

suppress

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
	gsl::suppress						

The [[gsl::suppress]] attribute suppresses specific clang-tidy diagnostics for rules of the C++ Core Guidelines in a portable way. The attribute can be attached to declarations, statements, and at namespace scope.

```
[[gsl::suppress("Rh-public")]]
void f_() {
  int *p;
  [[gsl::suppress("type")]] {
    p = reinterpret_cast<int*>(7);
  }
}
namespace N {
  [[clang::suppress("type", "bounds")]];
  ...
}
```

sycl_special_class

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
sycl_special_class	clang::sycl_special_class	clang::sycl_special_class					Yes

SYCL defines some special classes (accessor, sampler, and stream) which require specific handling during the generation of the SPIR entry point. The `__attribute__((sycl_special_class))` attribute is used in SYCL headers to indicate that a class or a struct needs a specific handling when it is passed from host to device. Special classes will have a mandatory `__init` method and an optional `__finalize` method (the `__finalize` method is used only with the `stream` type). Kernel parameters types are extract from the `__init` method parameters. The kernel function arguments list is derived from the arguments of the `__init` method. The arguments of the `__init` method are copied into the kernel function argument list and the `__init` and `__finalize` methods are called at the beginning and the end of the kernel, respectively. The `__init` and `__finalize` methods must be defined inside the special class. Please note that this is an attribute that is used as an internal implementation detail and not intended to be used by external users.

The syntax of the attribute is as follows:

```
class __attribute__((sycl_special_class)) accessor {};
class [[clang::sycl_special_class]] accessor {};
```

This is a code example that illustrates the use of the attribute:

```
class __attribute__((sycl_special_class)) SpecialType {
  int F1;
  int F2;
  void __init(int f1) {
    F1 = f1;
    F2 = f1;
  }
  void __finalize() {}
public:
  SpecialType() = default;
  int getF2() const { return F2; }
};

int main () {
  SpecialType T;
  cgh.single_task([=] {
    T.getF2();
  });
}
```

This would trigger the following kernel entry point in the AST:

```
void __sycl_kernel(int f1) {
  SpecialType T;
  T.__init(f1);
  ...
  T.__finalize()
}
```

Type Attributes

__ptr32

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
				__ptr32			

The `__ptr32` qualifier represents a native pointer on a 32-bit system. On a 64-bit system, a pointer with `__ptr32` is extended to a 64-bit pointer. The `__sptr` and `__uptr` qualifiers can be used to specify whether the pointer is sign extended or zero extended. This qualifier is enabled under `-fms-extensions`.

__ptr64

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
				__ptr64			

The `__ptr64` qualifier represents a native pointer on a 64-bit system. On a 32-bit system, a `__ptr64` pointer is truncated to a 32-bit pointer. This qualifier is enabled under `-fms-extensions`.

__sptr

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
				__sptr			

The `__sptr` qualifier specifies that a 32-bit pointer should be sign extended when converted to a 64-bit pointer.

__uptr

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
				__uptr			

The `__uptr` qualifier specifies that a 32-bit pointer should be zero extended when converted to a 64-bit pointer.

align_value

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
align_value							Yes

The align_value attribute can be added to the typedef of a pointer type or the declaration of a variable of pointer or reference type. It specifies that the pointer will point to, or the reference will bind to, only objects with at least the provided alignment. This alignment value must be some positive power of 2.

```
typedef double * aligned_double_ptr __attribute__((align_value(64)));
void foo(double & x  __attribute__((align_value(128)),
         aligned_double_ptr y) { ... }
```

If the pointer value does not have the specified alignment at runtime, the behavior of the program is undefined.

arm_sve_vector_bits

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
arm_sve_vector_bits							

The `arm_sve_vector_bits(N)` attribute is defined by the Arm C Language Extensions (ACLE) for SVE. It is used to define fixed-length (VLST) variants of sizeless types (VLAT).

For example:

```
#include <arm_sve.h>

#if __ARM_FEATURE_SVE_BITS==512
typedef svint32_t fixed_svint32_t __attribute__((arm_sve_vector_bits(512)));
#endif
```

Creates a type `fixed_svint32_t` that is a fixed-length variant of `svint32_t` that contains exactly 512-bits. Unlike `svint32_t`, this type can be used in globals, structs, unions, and arrays, all of which are unsupported for sizeless types.

The attribute can be attached to a single SVE vector (such as `svint32_t`) or to the SVE predicate type `svbool_t`, this excludes tuple types such as `svint32x4_t`. The behavior of the attribute is undefined unless `N==__ARM_FEATURE_SVE_BITS`, the implementation defined feature macro that is enabled under the `-msve-vector-bits` flag.

For more information See Arm C Language Extensions for SVE for more information.

btf_type_tag

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
btf_type_tag	clang::btf_type_tag	clang::btf_type_tag					

Clang supports the `__attribute__((btf_type_tag("ARGUMENT")))` attribute for all targets. It only has effect when `-g` is specified on the command line and is currently silently ignored when not applied to a pointer type (note: this scenario may be diagnosed in the future).

The `ARGUMENT` string will be preserved in IR and emitted to DWARF for the types used in variable declarations, function declarations, or typedef declarations.

For BPF targets, the `ARGUMENT` string will also be emitted to .BTF ELF section.

clang_arm_mve_strict_polymorphism

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
__clang_arm_mve_strict_polymorphism	clang::__clang_arm_mve_strict_polymorphism	clang::__clang_arm_mve_strict_polymorphism					

This attribute is used in the implementation of the ACLE intrinsics for the Arm MVE instruction set. It is used to define the vector types used by the MVE intrinsics.

Its effect is to modify the behavior of a vector type with respect to function overloading. If a candidate function for overload resolution has a parameter type with this attribute, then the selection of that candidate function will be disallowed if the actual argument can only be converted via a lax vector conversion. The aim is to prevent spurious ambiguity in ARM MVE polymorphic intrinsics.

```
void overloaded(uint16x8_t vector, uint16_t scalar);
void overloaded(int32x4_t vector, int32_t scalar);
uint16x8_t myVector;
uint16_t myScalar;

// myScalar is promoted to int32_t as a side effect of the addition,
// so if lax vector conversions are considered for myVector, then
// the two overloads are equally good (one argument conversion
// each). But if the vector has the __clang_arm_mve_strict_polymorphism
// attribute, only the uint16x8_t,uint16_t overload will match.
overloaded(myVector, myScalar + 1);
```

However, this attribute does not prohibit lax vector conversions in contexts other than overloading.

```
uint16x8_t function();

// This is still permitted with lax vector conversion enabled, even
// if the vector types have __clang_arm_mve_strict_polymorphism
int32x4_t result = function();
```

cmse_nonsecure_call

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
cmse_nonsecure_call							

This attribute declares a non-secure function type. When compiling for secure state, a call to such a function would switch from secure to non-secure state. All non-secure function calls must happen only through a function pointer, and a non-secure function type should only be used as a base type of a pointer. See ARMv8-M Security Extensions: Requirements on Development Tools - Engineering Specification Documentation for more information.

device_builtin_surface_type

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
device_builtin_surface_type			__device_builtin_surface_type__				Yes

The `device_builtin_surface_type` attribute can be applied to a class template when declaring the surface reference. A surface reference variable could be accessed on the host side and, on the device side, might be translated into an internal surface object, which is established through surface bind and unbind runtime APIs.

device_builtin_texture_type

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
device_builtin_texture_type			__device_builtin_texture_type__				Yes

The `device_builtin_texture_type` attribute can be applied to a class template when declaring the texture reference. A texture reference variable could be accessed on the host side and, on the device side, might be translated into an internal texture object, which is established through texture bind and unbind runtime APIs.

noderef

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
noderef	clang::noderef	clang::noderef					

The `noderef` attribute causes clang to diagnose dereferences of annotated pointer types. This is ideally used with pointers that point to special memory which cannot be read from or written to, but allowing for the pointer to be used in pointer arithmetic. The following are examples of valid expressions where dereferences are diagnosed:

```
int __attribute__((noderef)) *p;
int x = *p;  // warning

int __attribute__((noderef)) **p2;
x = **p2;  // warning

int * __attribute__((noderef)) *p3;
p = *p3;  // warning

struct S {
  int a;
};
struct S __attribute__((noderef)) *s;
x = s->a;    // warning
x = (*s).a;  // warning
```

Not all dereferences may diagnose a warning if the value directed by the pointer may not be accessed. The following are examples of valid expressions where may not be diagnosed:

```
int *q;
int __attribute__((noderef)) *p;
q = &*p;
q = *&p;

struct S {
  int a;
};
struct S __attribute__((noderef)) *s;
p = &s->a;
p = &(*s).a;
```

`noderef` is currently only supported for pointers and arrays and not usable for references or Objective-C object pointers.

objc_class_stub

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
objc_class_stub	clang::objc_class_stub	clang::objc_class_stub					Yes

This attribute specifies that the Objective-C class to which it applies is instantiated at runtime.

Unlike `__attribute__((objc_runtime_visible))`, a class having this attribute still has a “class stub” that is visible to the linker. This allows categories to be defined. Static message sends with the class as a receiver use a special access pattern to ensure the class is lazily instantiated from the class stub.

Classes annotated with this attribute cannot be subclassed and cannot have implementations defined for them. This attribute is intended for use in Swift-generated headers for classes defined in Swift.

Adding or removing this attribute to a class is an ABI-breaking change.

Type Safety Checking

Clang supports additional attributes to enable checking type safety properties that can’t be enforced by the C type system. To see warnings produced by these checks, ensure that -Wtype-safety is enabled. Use cases include:

- MPI library implementations, where these attributes enable checking that the buffer type matches the passed `MPI_Datatype`;
- for HDF5 library there is a similar use case to MPI;
- checking types of variadic functions’ arguments for functions like `fcntl()` and `ioctl()`.

You can detect support for these attributes with `__has_attribute()`. For example:

```
#if defined(__has_attribute)
#  if __has_attribute(argument_with_type_tag) && \
      __has_attribute(pointer_with_type_tag) && \
      __has_attribute(type_tag_for_datatype)
#    define ATTR_MPI_PWT(buffer_idx, type_idx) __attribute__((pointer_with_type_tag(mpi,buffer_idx,type_idx)))
/* ... other macros ... */
#  endif
#endif
```

```
#if !defined(ATTR_MPI_PWT)
# define ATTR_MPI_PWT(buffer_idx, type_idx)
#endif

int MPI_Send(void *buf, int count, MPI_Datatype datatype /*, other args omitted */)
    ATTR_MPI_PWT(1,3);
```

argument_with_type_tag

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
argument_with_type_tag pointer_with_type_tag	clang::argument_with_type_tag clang::pointer_with_type_tag	clang::argument_with_type_tag clang::pointer_with_type_tag					

Use `__attribute__((argument_with_type_tag(arg_kind, arg_idx, type_tag_idx)))` on a function declaration to specify that the function accepts a type tag that determines the type of some other argument.

This attribute is primarily useful for checking arguments of variadic functions (`pointer_with_type_tag` can be used in most non-variadic cases).

In the attribute prototype above:

- `arg_kind` is an identifier that should be used when annotating all applicable type tags.
- `arg_idx` provides the position of a function argument. The expected type of this function argument will be determined by the function argument specified by `type_tag_idx`. In the code example below, “3” means that the type of the function’s third argument will be determined by `type_tag_idx`.
- `type_tag_idx` provides the position of a function argument. This function argument will be a type tag. The type tag will determine the expected type of the argument specified by `arg_idx`. In the code example below, “2” means that the type tag associated with the function’s second argument should agree with the type of the argument specified by `arg_idx`.

For example:

```
int fcntl(int fd, int cmd, ...)
    __attribute__(( argument_with_type_tag(fcntl,3,2) ));
// The function's second argument will be a type tag; this type tag will
// determine the expected type of the function's third argument.
```

pointer_with_type_tag

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
argument_with_type_tag pointer_with_type_tag	clang::argument_with_type_tag clang::pointer_with_type_tag	clang::argument_with_type_tag clang::pointer_with_type_tag					

Use `__attribute__((pointer_with_type_tag(ptr_kind, ptr_idx, type_tag_idx)))` on a function declaration to specify that the function accepts a type tag that determines the pointee type of some other pointer argument.

In the attribute prototype above:

- `ptr_kind` is an identifier that should be used when annotating all applicable type tags.
- `ptr_idx` provides the position of a function argument; this function argument will have a pointer type. The expected pointee type of this pointer type will be determined by the function argument specified by `type_tag_idx`. In the code example below, “1” means that the pointee type of the function’s first argument will be determined by `type_tag_idx`.
- `type_tag_idx` provides the position of a function argument; this function argument will be a type tag. The type tag will determine the expected pointee type of the pointer argument specified by `ptr_idx`. In the code example below, “3” means that the type tag associated with the function’s third argument should agree with the pointee type of the pointer argument specified by `ptr_idx`.

For example:

```
typedef int MPI_Datatype;
int MPI_Send(void *buf, int count, MPI_Datatype datatype /*, other args omitted */)
    __attribute__(( pointer_with_type_tag(mpi,1,3) ));
// The function's 3rd argument will be a type tag; this type tag will
// determine the expected pointee type of the function's 1st argument.
```

type_tag_for_datatype

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
type_tag_for_datatype	clang::type_tag_for_datatype	clang::type_tag_for_datatype					

When declaring a variable, use `__attribute__((type_tag_for_datatype(kind, type)))` to create a type tag that is tied to the `type` argument given to the attribute.

In the attribute prototype above:

- `kind` is an identifier that should be used when annotating all applicable type tags.
- `type` indicates the name of the type.

Clang supports annotating type tags of two forms.

- **Type tag that is a reference to a declared identifier.** Use `__attribute__((type_tag_for_datatype(kind, type)))` when declaring that identifier:

```
typedef int MPI_Datatype;
extern struct mpi_datatype mpi_datatype_int
    __attribute__(( type_tag_for_datatype(mpi,int) ));
#define MPI_INT ((MPI_Datatype) &mpi_datatype_int)
// &mpi_datatype_int is a type tag. It is tied to type "int".
```

- **Type tag that is an integral literal.** Declare a `static const` variable with an initializer value and attach `__attribute__((type_tag_for_datatype(kind, type)))` on that declaration:

```
typedef int MPI_Datatype;
static const MPI_Datatype mpi_datatype_int
    __attribute__(( type_tag_for_datatype(mpi,int) )) = 42;
#define MPI_INT ((MPI_Datatype) 42)
// The number 42 is a type tag. It is tied to type "int".
```

The `type_tag_for_datatype` attribute also accepts an optional third argument that determines how the type of the function argument specified by either `arg_idx` or `ptr_idx` is compared against the type associated with the type tag. (Recall that for the `argument_with_type_tag` attribute, the type of the function argument specified by `arg_idx` is compared against the type associated with the type tag. Also recall that for the `pointer_with_type_tag` attribute, the pointee type of the function argument specified by `ptr_idx` is compared against the type associated with the type tag.) There are two supported values for this optional third argument:

- `layout_compatible` will cause types to be compared according to layout-compatibility rules (In C++11 [class.mem] p 17, 18, see the layout-compatibility rules for two standard-layout struct types and for two standard-layout union types). This is useful when creating a type tag associated with a struct or union type. For example:

```
/* In mpi.h */
typedef int MPI_Datatype;
struct internal_mpi_double_int { double d; int i; };
extern struct mpi_datatype mpi_datatype_double_int
    __attribute__(( type_tag_for_datatype(mpi,
                    struct internal_mpi_double_int, layout_compatible) ));

#define MPI_DOUBLE_INT ((MPI_Datatype) &mpi_datatype_double_int)

int MPI_Send(void *buf, int count, MPI_Datatype datatype, ...)
    __attribute__(( pointer_with_type_tag(mpi,1,3) ));

/* In user code */
struct my_pair { double a; int b; };
struct my_pair *buffer;
MPI_Send(buffer, 1, MPI_DOUBLE_INT /*, ... */); // no warning because the
                                                // layout of my_pair is
                                                // compatible with that of
                                                // internal_mpi_double_int

struct my_int_pair { int a; int b; }
struct my_int_pair *buffer2;
MPI_Send(buffer2, 1, MPI_DOUBLE_INT /*, ... */); // warning because the
                                                 // layout of my_int_pair
                                                 // does not match that of
                                                 // internal_mpi_double_int
```

- `must_be_null` specifies that the function argument specified by either `arg_idx` (for the `argument_with_type_tag` attribute) or `ptr_idx` (for the `pointer_with_type_tag` attribute) should be a null pointer constant. The second argument to the `type_tag_for_datatype` attribute is ignored. For example:

```
/* In mpi.h */
typedef int MPI_Datatype;
extern struct mpi_datatype mpi_datatype_null
```

```
    __attribute__(( type_tag_for_datatype(mpi, void, must_be_null) ));

#define MPI_DATATYPE_NULL ((MPI_Datatype) &mpi_datatype_null)
int MPI_Send(void *buf, int count, MPI_Datatype datatype, ...)
    __attribute__(( pointer_with_type_tag(mpi,1,3) ));

/* In user code */
struct my_pair { double a; int b; };
struct my_pair *buffer;
MPI_Send(buffer, 1, MPI_DATATYPE_NULL /*, ... */); // warning: MPI_DATATYPE_NULL
                                                   // was specified but buffer
                                                   // is not a null pointer
```

Variable Attributes

always_destroy

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
always_destroy	clang::always_destroy						Yes

The `always_destroy` attribute specifies that a variable with static or thread storage duration should have its exit-time destructor run. This attribute is the default unless clang was invoked with -fno-c++-static-destructors.

called_once

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
called_once	clang::called_once	clang::called_once					Yes

The `called_once` attribute specifies that the annotated function or method parameter is invoked exactly once on all execution paths. It only applies to parameters with function-like types, i.e. function pointers or blocks. This concept is particularly useful for asynchronous programs.

Clang implements a check for `called_once` parameters, `-Wcalled-once-parameter`. It is on by default and finds the following violations:

- Parameter is not called at all.
- Parameter is called more than once.
- Parameter is not called on one of the execution paths.

In the latter case, Clang pinpoints the path where parameter is not invoked by showing the control-flow statement where the path diverges.

```
void fooWithCallback(void (^callback)(void) __attribute__((called_once))) {
  if (somePredicate()) {
    ...
```

```
    callback();
  } esle {
    callback(); // OK: callback is called on every path
  }
}

void barWithCallback(void (^callback)(void) __attribute__((called_once))) {
  if (somePredicate()) {
    ...
    callback(); // note: previous call is here
  }
  callback(); // warning: callback is called twice
}

void foobarWithCallback(void (^callback)(void) __attribute__((called_once))) {
  if (somePredicate()) {  // warning: callback is not called when condition is false
    ...
    callback();
  }
}
```

This attribute is useful for API developers who want to double-check if they implemented their method correctly.

dllexport

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
dllexport	gnu::dllexport	gnu::dllexport	dllexport				Yes

The `__declspec(dllexport)` attribute declares a variable, function, or Objective-C interface to be exported from the module. It is available under the `-fdeclspec` flag for compatibility with various compilers. The primary use is for COFF object files which explicitly specify what interfaces are available for external use. See the dllexport documentation on MSDN for more information.

dllimport

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
dllimport	gnu::dllimport	gnu::dllimport	dllimport				Yes

The `__declspec(dllimport)` attribute declares a variable, function, or Objective-C interface to be imported from an external module. It is available under the `-fdeclspec` flag for compatibility with various compilers. The primary use is for COFF object files which explicitly specify what interfaces are imported from external modules. See the dllimport documentation on MSDN for more information.

Note that a dllimport function may still be inlined, if its definition is available and it doesn't reference any non-dllimport functions or global variables.

init_priority

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
init_priority	gnu::init_priority						Yes

In C++, the order in which global variables are initialized across translation units is unspecified, unlike the ordering within a single translation unit. The `init_priority` attribute allows you to specify a relative ordering for the initialization of objects declared at namespace scope in C++. The priority is given as an integer constant expression between 101 and 65535 (inclusive). Priorities outside of that range are reserved for use by the implementation. A lower value indicates a higher priority of initialization. Note that only the relative ordering of values is important. For example:

```
struct SomeType { SomeType(); };
__attribute__((init_priority(200))) SomeType Obj1;
__attribute__((init_priority(101))) SomeType Obj2;
```

`Obj2` will be initialized *before* `Obj1` despite the usual order of initialization being the opposite.

This attribute is only supported for C++ and Objective-C++ and is ignored in other language modes. Currently, this attribute is not implemented on z/OS.

init_seg

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
					init_seg		

The attribute applied by `pragma init_seg()` controls the section into which global initialization function pointers are emitted. It is only available with `-fms-extensions`. Typically, this function pointer is emitted into `.CRT$XCU` on Windows. The user can change the order of initialization by using a different section name with the same `.CRT$XC` prefix and a suffix that sorts lexicographically before or after the standard `.CRT$XCU` sections. See the init_seg documentation on MSDN for more information.

leaf

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
leaf	gnu::leaf	gnu::leaf					Yes

The `leaf` attribute is used as a compiler hint to improve dataflow analysis in library functions. Functions marked with the `leaf` attribute are not allowed to jump back into the caller's translation unit, whether through invoking a callback function, an external function call, use of `longjmp`, or other means. Therefore, they cannot use or modify any data that does not escape the caller function's compilation unit.

For more information see *gcc documentation <https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html>*

loader_uninitialized

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
loader_uninitialized	clang::loader_uninitialized	clang::loader_uninitialized					Yes

The `loader_uninitialized` attribute can be placed on global variables to indicate that the variable does not need to be zero initialized by the loader. On most targets, zero-initialization does not incur any additional cost. For example, most general purpose operating systems deliberately ensure that all memory is properly initialized in order to avoid leaking privileged information from the kernel or other programs. However, some targets do not make this guarantee, and on these targets, avoiding an unnecessary zero-initialization can have a significant impact on load times and/or code size.

A declaration with this attribute is a non-tentative definition just as if it provided an initializer. Variables with this attribute are considered to be uninitialized in the same sense as a local variable, and the programs must write to them before reading from them. If the variable's type is a C++ class type with a non-trivial default constructor, or an array thereof, this attribute only suppresses the static zero-initialization of the variable, not the dynamic initialization provided by executing the default constructor.

maybe_unused, unused

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
unused	maybe_unused gnu::unused	gnu::unused maybe_unused					

When passing the `-Wunused` flag to Clang, entities that are unused by the program may be diagnosed. The `[[maybe_unused]]` (or `__attribute__((unused))`) attribute can be used to silence such diagnostics when the entity cannot be removed. For instance, a local variable may exist solely for use in an `assert()` statement, which makes the local variable unused when `NDEBUG` is defined.

The attribute may be applied to the declaration of a class, a typedef, a variable, a function or method, a function parameter, an enumeration, an enumerator, a non-static data member, or a label.

no_destroy

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
no_destroy	clang::no_destroy						Yes

The no_destroy attribute specifies that a variable with static or thread storage duration shouldn't have its exit-time destructor run. Annotating every static and thread duration variable with this attribute is equivalent to invoking clang with -fno-c++-static-destructors.

If a variable is declared with this attribute, clang doesn't access check or generate the type's destructor. If you have a type that you only want to be annotated with no_destroy, you can therefore declare the destructor private:

```
struct only_no_destroy {
  only_no_destroy();
private:
  ~only_no_destroy();
};

[[clang::no_destroy]] only_no_destroy global; // fine!
```

Note that destructors are still required for subobjects of aggregates annotated with this attribute. This is because previously constructed subobjects need to be destroyed if an exception gets thrown before the initialization of the complete object is complete. For instance:

```
void f() {
  try {
    [[clang::no_destroy]]
    static only_no_destroy array[10]; // error, only_no_destroy has a private destructor.
  } catch (...) {
    // Handle the error
  }
}
```

Here, if the construction of array[9] fails with an exception, array[0..8] will be destroyed, so the element's destructor needs to be accessible.

nodebug

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
nodebug	gnu::nodebug	gnu::nodebug					Yes

The nodebug attribute allows you to suppress debugging information for a function or method, for a variable that is not a parameter or a non-static data member, or for a typedef or using declaration.

noescape

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
noescape	clang::noescape	clang::noescape					Yes

`noescape` placed on a function parameter of a pointer type is used to inform the compiler that the pointer cannot escape: that is, no reference to the object the pointer points to that is derived from the parameter value will survive after the function returns. Users are responsible for making sure parameters annotated with `noescape` do not actually escape. Calling `free()` on such a parameter does not constitute an escape.

For example:

```
int *gp;

void nonescapingFunc(__attribute__((noescape)) int *p) {
  *p += 100; // OK.
}

void escapingFunc(__attribute__((noescape)) int *p) {
  gp = p; // Not OK.
}
```

Additionally, when the parameter is a *block pointer <https://clang.llvm.org/docs/BlockLanguageSpec.html>*, the same restriction applies to copies of the block. For example:

```
typedef void (^BlockTy)();
BlockTy g0, g1;

void nonescapingFunc(__attribute__((noescape)) BlockTy block) {
  block(); // OK.
}

void escapingFunc(__attribute__((noescape)) BlockTy block) {
  g0 = block; // Not OK.
  g1 = Block_copy(block); // Not OK either.
}
```

nosvm

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
nosvm							Yes

OpenCL 2.0 supports the optional `__attribute__((nosvm))` qualifier for pointer variable. It informs the compiler that the pointer does not refer to a shared virtual memory region. See OpenCL v2.0 s6.7.2 for details.

Since it is not widely used and has been removed from OpenCL 2.1, it is ignored by Clang.

objc_externally_retained

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
objc_externally_retained	clang::objc_externally_retained	clang::objc_externally_retained					Yes

The `objc_externally_retained` attribute can be applied to strong local variables, functions, methods, or blocks to opt into externally-retained semantics.

When applied to the definition of a function, method, or block, every parameter of the function with implicit strong retainable object pointer type is considered externally-retained, and becomes `const`. By explicitly annotating a parameter with `__strong`, you can opt back into the default non-externally-retained behavior for that parameter. For instance, `first_param` is externally-retained below, but not `second_param`:

```
__attribute__((objc_externally_retained))
void f(NSArray *first_param, __strong NSArray *second_param) {
  // ...
}
```

Likewise, when applied to a strong local variable, that variable becomes `const` and is considered externally-retained.

When compiled without `-fobjc-arc`, this attribute is ignored.

pass_object_size, pass_dynamic_object_size

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
pass_object_size pass_dynamic_object_size	clang::pass_object_size clang::pass_dynamic_object_size	clang::pass_object_size clang::pass_dynamic_object_size					Yes

Note

The mangling of functions with parameters that are annotated with `pass_object_size` is subject to change. You can get around this by using `__asm__("foo")` to explicitly name your functions, thus preserving your ABI; also, non-overloadable C functions with `pass_object_size` are not mangled.

The `pass_object_size(Type)` attribute can be placed on function parameters to instruct clang to call `__builtin_object_size(param, Type)` at each callsite of said function, and implicitly pass the result of this call in as an invisible argument of type `size_t` directly after the parameter annotated with `pass_object_size`. Clang will also replace any calls to `__builtin_object_size(param, Type)` in the function by said implicit parameter.

Example usage:

```
int bzero1(char *const p __attribute__((pass_object_size(0))))
    __attribute__((noinline)) {
  int i = 0;
```

```
  for (/**/; i < (int)__builtin_object_size(p, 0); ++i) {
    p[i] = 0;
  }
  return i;
}

int main() {
  char chars[100];
  int n = bzero1(&chars[0]);
  assert(n == sizeof(chars));
  return 0;
}
```

If successfully evaluating `__builtin_object_size(param, Type)` at the callsite is not possible, then the "failed" value is passed in. So, using the definition of `bzero1` from above, the following code would exit cleanly:

```
int main2(int argc, char *argv[]) {
  int n = bzero1(argv);
  assert(n == -1);
  return 0;
}
```

`pass_object_size` plays a part in overload resolution. If two overload candidates are otherwise equally good, then the overload with one or more parameters with `pass_object_size` is preferred. This implies that the choice between two identical overloads both with `pass_object_size` on one or more parameters will always be ambiguous; for this reason, having two such overloads is illegal. For example:

```
#define PS(N) __attribute__((pass_object_size(N)))
// OK
void Foo(char *a, char *b); // Overload A
// OK -- overload A has no parameters with pass_object_size.
void Foo(char *a PS(0), char *b PS(0)); // Overload B
// Error -- Same signature (sans pass_object_size) as overload B, and both
// overloads have one or more parameters with the pass_object_size attribute.
void Foo(void *a PS(0), void *b);

// OK
void Bar(void *a PS(0)); // Overload C
// OK
void Bar(char *c PS(1)); // Overload D

void main() {
  char known[10], *unknown;
  Foo(unknown, unknown); // Calls overload B
  Foo(known, unknown); // Calls overload B
  Foo(unknown, known); // Calls overload B
  Foo(known, known); // Calls overload B

  Bar(known); // Calls overload D
  Bar(unknown); // Calls overload D
}
```

Currently, `pass_object_size` is a bit restricted in terms of its usage:

- Only one use of `pass_object_size` is allowed per parameter.
- It is an error to take the address of a function with `pass_object_size` on any of its parameters. If you wish to do this, you can create an overload without `pass_object_size` on any parameters.
- It is an error to apply the `pass_object_size` attribute to parameters that are not pointers. Additionally, any parameter that `pass_object_size` is applied to must be marked `const` at its function's definition.

Clang also supports the `pass_dynamic_object_size` attribute, which behaves identically to `pass_object_size`, but evaluates a call to `__builtin_dynamic_object_size` at the callee instead of

`__builtin_object_size`. `__builtin_dynamic_object_size` provides some extra runtime checks when the object size can't be determined at compile-time. You can read more about `__builtin_dynamic_object_size` here.

require_constant_initialization, constinit (C++20)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
require_constant_initialization	clang::require_constant_initialization			constinit			Yes

This attribute specifies that the variable to which it is attached is intended to have a constant initializer according to the rules of [basic.start.static]. The variable is required to have static or thread storage duration. If the initialization of the variable is not a constant initializer an error will be produced. This attribute may only be used in C++; the `constinit` spelling is only accepted in C++20 onwards.

Note that in C++03 strict constant expression checking is not done. Instead the attribute reports if Clang can emit the variable as a constant, even if it's not technically a 'constant initializer'. This behavior is non-portable.

Static storage duration variables with constant initializers avoid hard-to-find bugs caused by the indeterminate order of dynamic initialization. They can also be safely used during dynamic initialization across translation units.

This attribute acts as a compile time assertion that the requirements for constant initialization have been met. Since these requirements change between dialects and have subtle pitfalls it's important to fail fast instead of silently falling back on dynamic initialization.

The first use of the attribute on a variable must be part of, or precede, the initializing declaration of the variable. C++20 requires the `constinit` spelling of the attribute to be present on the initializing declaration if it is used anywhere. The other spellings can be specified on a forward declaration and omitted on a later initializing declaration.

```
// -std=c++14
#define SAFE_STATIC [[clang::require_constant_initialization]]
struct T {
  constexpr T(int) {}
  ~T(); // non-trivial
};
SAFE_STATIC T x = {42}; // Initialization OK. Doesn't check destructor.
SAFE_STATIC T y = 42; // error: variable does not have a constant initializer
// copy initialization is not a constant expression on a non-literal type.
```

section, __declspec(allocate)

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
section	gnu::section	gnu::section	allocate				Yes

The `section` attribute allows you to specify a specific section a global variable or function should be in after translation.

standalone_debug

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
standalone_debug	clang::standalone_debug						Yes

The `standalone_debug` attribute causes debug info to be emitted for a record type regardless of the debug info optimizations that are enabled with -fno-standalone-debug. This attribute only has an effect when debug info optimizations are enabled (e.g. with -fno-standalone-debug), and is C++-only.

swift_async_context

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
swift_async_context	clang::swift_async_context	clang::swift_async_context					Yes

The `swift_async_context` attribute marks a parameter of a `swiftasynccall` function as having the special asynchronous context-parameter ABI treatment.

If the function is not `swiftasynccall`, this attribute only generates extended frame information.

A context parameter must have pointer or reference type.

swift_context

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
swift_context	clang::swift_context	clang::swift_context					Yes

The `swift_context` attribute marks a parameter of a `swiftcall` or `swiftasynccall` function as having the special context-parameter ABI treatment.

This treatment generally passes the context value in a special register which is normally callee-preserved.

A `swift_context` parameter must either be the last parameter or must be followed by a `swift_error_result` parameter (which itself must always be the last parameter).

A context parameter must have pointer or reference type.

swift_error_result

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
swift_error_result	clang::swift_error_result	clang::swift_error_result					Yes

The `swift_error_result` attribute marks a parameter of a `swiftcall` function as having the special error-result ABI treatment.

This treatment generally passes the underlying error value in and out of the function through a special register which is normally callee-preserved. This is modeled in C by pretending that the register is addressable memory:

- The caller appears to pass the address of a variable of pointer type. The current value of this variable is copied into the register before the call; if the call returns normally, the value is copied back into the variable.
- The callee appears to receive the address of a variable. This address is actually a hidden location in its own stack, initialized with the value of the register upon entry. When the function returns normally, the value in that hidden location is written back to the register.

A `swift_error_result` parameter must be the last parameter, and it must be preceded by a `swift_context` parameter.

A `swift_error_result` parameter must have type `T**` or `T*&` for some type `T`. Note that no qualifiers are permitted on the intermediate level.

It is undefined behavior if the caller does not pass a pointer or reference to a valid object.

The standard convention is that the error value itself (that is, the value stored in the apparent argument) will be null upon function entry, but this is not enforced by the ABI.

swift_indirect_result

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
swift_indirect_result	clang::swift_indirect_result	clang::swift_indirect_result					Yes

The `swift_indirect_result` attribute marks a parameter of a `swiftcall` or `swiftasynccall` function as having the special indirect-result ABI treatment.

This treatment gives the parameter the target's normal indirect-result ABI treatment, which may involve passing it differently from an ordinary parameter. However, only the first indirect result will receive this treatment. Furthermore, low-level lowering may decide that a direct result must be returned indirectly; if so, this will take priority over the `swift_indirect_result` parameters.

A `swift_indirect_result` parameter must either be the first parameter or follow another `swift_indirect_result` parameter.

A `swift_indirect_result` parameter must have type `T*` or `T&` for some object type `T`. If `T` is a complete type at the point of definition of a function, it is undefined behavior if the argument value does not point to storage of adequate size and alignment for a value of type `T`.

Making indirect results explicit in the signature allows C functions to directly construct objects into them without relying on language optimizations like C++'s named return value optimization (NRVO).

swiftasynccall

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
swiftasynccall	clang::swiftasynccall	clang::swiftasynccall					

The `swiftasynccall` attribute indicates that a function is compatible with the low-level conventions of Swift async functions, provided it declares the right formal arguments.

In most respects, this is similar to the `swiftcall` attribute, except for the following:

- A parameter may be marked `swift_async_context`, `swift_context` or `swift_indirect_result` (with the same restrictions on parameter ordering as `swiftcall`) but the parameter attribute `swift_error_result` is not permitted.
- A `swiftasynccall` function must have return type `void`.
- Within a `swiftasynccall` function, a call to a `swiftasynccall` function that is the immediate operand of a `return` statement is guaranteed to be performed as a tail call. This syntax is allowed even in C as an extension (a call to a void-returning function cannot be a return operand in standard C). If something in the calling function would semantically be performed after a guaranteed tail call, such as the non-trivial destruction of a local variable or temporary, then the program is ill-formed.

swiftcall

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
swiftcall	clang::swiftcall	clang::swiftcall					

The `swiftcall` attribute indicates that a function should be called using the Swift calling convention for a function or function pointer.

The lowering for the Swift calling convention, as described by the Swift ABI documentation, occurs in multiple phases. The first, “high-level” phase breaks down the formal parameters and results into innately direct and indirect components, adds implicit parameters for the generic signature, and assigns the context and error ABI treatments to parameters where applicable. The second phase breaks down the direct parameters and results from the first phase and assigns them to registers or the stack. The `swiftcall` convention only handles this second phase of lowering; the C function type must accurately reflect the results of the first phase, as follows:

- Results classified as indirect by high-level lowering should be represented as parameters with the `swift_indirect_result` attribute.
- Results classified as direct by high-level lowering should be represented as follows:
  - First, remove any empty direct results.
  - If there are no direct results, the C result type should be `void`.
  - If there is one direct result, the C result type should be a type with the exact layout of that result type.
  - If there are a multiple direct results, the C result type should be a struct type with the exact layout of a tuple of those results.
- Parameters classified as indirect by high-level lowering should be represented as parameters of pointer type.

- Parameters classified as direct by high-level lowering should be omitted if they are empty types; otherwise, they should be represented as a parameter type with a layout exactly matching the layout of the Swift parameter type.
- The context parameter, if present, should be represented as a trailing parameter with the `swift_context` attribute.
- The error result parameter, if present, should be represented as a trailing parameter (always following a context parameter) with the `swift_error_result` attribute.

`swiftcall` does not support variadic arguments or unprototyped functions.

The parameter ABI treatment attributes are aspects of the function type. A function type which applies an ABI treatment attribute to a parameter is a different type from an otherwise-identical function type that does not. A single parameter may not have multiple ABI treatment attributes.

Support for this feature is target-dependent, although it should be supported on every target that Swift supports. Query for this support with `__has_attribute(swiftcall)`. This implies support for the `swift_context`, `swift_error_result`, and `swift_indirect_result` attributes.

thread

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
			thread				

The `__declspec(thread)` attribute declares a variable with thread local storage. It is available under the `-fms-extensions` flag for MSVC compatibility. See the documentation for __declspec(thread) on MSDN.

In Clang, `__declspec(thread)` is generally equivalent in functionality to the GNU `__thread` keyword. The variable must not have a destructor and must have a constant initializer, if any. The attribute only applies to variables declared with static storage duration, such as globals, class static data members, and static locals.

tls_model

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
tls_model	gnu::tls_model	gnu::tls_model					Yes

The `tls_model` attribute allows you to specify which thread-local storage model to use. It accepts the following strings:

- global-dynamic
- local-dynamic
- initial-exec
- local-exec

TLS models are mutually exclusive.

uninitialized

Supported Syntaxes

GNU	C++11	C2x	__declspec	Keyword	#pragma	#pragma clang attribute	HLSL Semantic
uninitialized	clang::uninitialized						Yes

The command-line parameter `-ftrivial-auto-var-init=*` can be used to initialize trivial automatic stack variables. By default, trivial automatic stack variables are uninitialized. This attribute is used to override the command-line parameter, forcing variables to remain uninitialized. It has no semantic meaning in that using uninitialized values is undefined behavior, it rather documents the programmer's intent.

Diagnostic flags in Clang

Introduction 396

Diagnostic flags 396

-W 396
-W#pragma-messages 396
-W#warnings 396
-WCFString-literal 396
-WCL4 396
-WIndependentClass-attribute 396
-WNSObject-attribute 396
-Wabi 397
-Wabsolute-value 397
-Wabstract-final-class 397
-Wabstract-vbase-init 397
-Waddress 397
-Waddress-of-packed-member 397
-Waddress-of-temporary 397
-Waggregate-return 397
-Waggressive-restrict 397
-Waix-compat 398
-Walign-mismatch 398
-Wall 398
-Walloca 398
-Walloca-with-align-alignof 398
-Walways-inline-coroutine 398
-Wambiguous-delete 398
-Wambiguous-ellipsis 398
-Wambiguous-macro 399
-Wambiguous-member-template 399
-Wambiguous-reversed-operator 399
-Wanalyzer-incompatible-plugin 399
-Wanon-enum-enum-conversion 399
-Wanonymous-pack-parens 399
-Warc 399
-Warc-bridge-casts-disallowed-in-nonarc 399
-Warc-maybe-repeated-use-of-weak 400
-Warc-non-pod-memaccess 400
-Warc-performSelector-leaks 400
-Warc-repeated-use-of-weak 400
-Warc-retain-cycles 400
-Warc-unsafe-retained-assign 400
-Wargument-outside-range 400
-Wargument-undefined-behaviour 400
-Warray-bounds 401

-Warray-bounds-pointer-arithmetic 401
-Wasm 401
-Wasm-operand-widths 401
-Wassign-enum 401
-Wassume 401
-Wat-protocol 401
-Watimport-in-framework-header 401
-Watomic-access 402
-Watomic-alignment 402
-Watomic-implicit-seq-cst 402
-Watomic-memory-ordering 402
-Watomic-properties 402
-Watomic-property-with-user-defined-accessor 402
-Wattribute-packed-for-bitfield 402
-Wattribute-warning 402
-Wattributes 403
-Wauto-disable-vptr-sanitizer 403
-Wauto-import 403
-Wauto-storage-class 403
-Wauto-var-id 403
-Wavailability 403
-Wavr-rtlib-linking-quirks 404
-Wbackend-plugin 404
-Wbackslash-newline-escape 404
-Wbad-function-cast 404
-Wbinary-literal 404
-Wbind-to-temporary-copy 404
-Wbinding-in-condition 404
-Wbit-int-extension 405
-Wbitfield-constant-conversion 405
-Wbitfield-enum-conversion 405
-Wbitfield-width 405
-Wbitwise-conditional-parentheses 405
-Wbitwise-instead-of-logical 405
-Wbitwise-op-parentheses 405
-Wblock-capture-autoreleasing 405
-Wbool-conversion 405
-Wbool-conversions 406
-Wbool-operation 406
-Wbraced-scalar-init 406
-Wbranch-protection 406
-Wbridge-cast 406
-Wbuiltin-assume-aligned-alignment 406
-Wbuiltin-macro-redefined 406
-Wbuiltin-memcpy-chk-size 407

-Wbuiltin-requires-header 407
-Wc++-compat 407
-Wc++0x-compat 407
-Wc++0x-extensions 407
-Wc++0x-narrowing 407
-Wc++11-compat 407
-Wc++11-compat-deprecated-writable-strings 408
-Wc++11-compat-pedantic 408
-Wc++11-compat-reserved-user-defined-literal 408
-Wc++11-extensions 408
-Wc++11-extra-semi 409
-Wc++11-inline-namespace 409
-Wc++11-long-long 409
-Wc++11-narrowing 409
-Wc++14-attribute-extensions 410
-Wc++14-binary-literal 410
-Wc++14-compat 410
-Wc++14-compat-pedantic 410
-Wc++14-extensions 410
-Wc++17-attribute-extensions 410
-Wc++17-compat 410
-Wc++17-compat-mangling 410
-Wc++17-compat-pedantic 411
-Wc++17-extensions 411
-Wc++1y-extensions 411
-Wc++1z-compat 411
-Wc++1z-compat-mangling 411
-Wc++1z-extensions 412
-Wc++20-attribute-extensions 412
-Wc++20-compat 412
-Wc++20-compat-pedantic 412
-Wc++20-designator 412
-Wc++20-extensions 412
-Wc++2a-compat 413
-Wc++2a-compat-pedantic 413
-Wc++2a-extensions 413
-Wc++2b-extensions 413
-Wc++98-c++11-c++14-c++17-compat 414
-Wc++98-c++11-c++14-c++17-compat-pedantic 414
-Wc++98-c++11-c++14-compat 414
-Wc++98-c++11-c++14-compat-pedantic 414
-Wc++98-c++11-compat 414
-Wc++98-c++11-compat-binary-literal 414
-Wc++98-c++11-compat-pedantic 414
-Wc++98-compat 414

-Wc++98-compat-bind-to-temporary-copy 416
-Wc++98-compat-extra-semi 416
-Wc++98-compat-local-type-template-args 416
-Wc++98-compat-pedantic 416
-Wc++98-compat-unnamed-type-template-args 417
-Wc11-extensions 417
-Wc2x-extensions 417
-Wc99-compat 417
-Wc99-designator 418
-Wc99-extensions 418
-Wcall-to-pure-virtual-from-ctor-dtor 418
-Wcalled-once-parameter 418
-Wcast-align 419
-Wcast-calling-convention 419
-Wcast-function-type 419
-Wcast-of-sel-type 419
-Wcast-qual 419
-Wcast-qual-unrelated 419
-Wchar-align 419
-Wchar-subscripts 419
-Wclang-cl-pch 420
-Wclass-conversion 420
-Wclass-varargs 420
-Wcmse-union-leak 420
-Wcomma 420
-Wcomment 420
-Wcomments 421
-Wcompare-distinct-pointer-types 421
-Wcompletion-handler 421
-Wcomplex-component-init 421
-Wcompound-token-split 421
-Wcompound-token-split-by-macro 421
-Wcompound-token-split-by-space 421
-Wconcepts-ts-compat 421
-Wconditional-type-mismatch 421
-Wconditional-uninitialized 422
-Wconfig-macros 422
-Wconstant-conversion 422
-Wconstant-evaluated 422
-Wconstant-logical-operand 422
-Wconstexpr-not-const 422
-Wconsumed 422
-Wconversion 423
-Wconversion-null 423
-Wcoroutine 423

-Wcoroutine-missing-unhandled-exception 423
-Wcovered-switch-default 423
-Wcpp 423
-Wcstring-format-directive 423
-Wctad-maybe-unsupported 424
-Wctor-dtor-privacy 424
-Wctu 424
-Wcuda-compat 424
-Wcustom-atomic-properties 424
-Wcxx-attribute-extension 424
-Wdangling 424
-Wdangling-else 424
-Wdangling-field 425
-Wdangling-gsl 425
-Wdangling-initializer-list 425
-Wdarwin-sdk-settings 425
-Wdate-time 425
-Wdealloc-in-category 425
-Wdebug-compression-unavailable 425
-Wdeclaration-after-statement 426
-Wdefaulted-function-deleted 426
-Wdelegating-ctor-cycles 426
-Wdelete-abstract-non-virtual-dtor 426
-Wdelete-incomplete 426
-Wdelete-non-abstract-non-virtual-dtor 426
-Wdelete-non-virtual-dtor 426
-Wdelimited-escape-sequence-extension 426
-Wdeprecate-lax-vec-conv-all 427
-Wdeprecated 427
-Wdeprecated-altivec-src-compat 427
-Wdeprecated-anon-enum-enum-conversion 427
-Wdeprecated-array-compare 427
-Wdeprecated-attributes 428
-Wdeprecated-comma-subscript 428
-Wdeprecated-copy 428
-Wdeprecated-copy-dtor 428
-Wdeprecated-copy-with-dtor 428
-Wdeprecated-copy-with-user-provided-copy 428
-Wdeprecated-copy-with-user-provided-dtor 428
-Wdeprecated-coroutine 428
-Wdeprecated-declarations 428
-Wdeprecated-dynamic-exception-spec 429
-Wdeprecated-enum-compare 429
-Wdeprecated-enum-compare-conditional 429
-Wdeprecated-enum-enum-conversion 429

-Wdeprecated-enum-float-conversion 429
-Wdeprecated-experimental-coroutine 429
-Wdeprecated-implementations 429
-Wdeprecated-increment-bool 430
-Wdeprecated-non-prototype 430
-Wdeprecated-objc-isa-usage 430
-Wdeprecated-objc-pointer-introspection 430
-Wdeprecated-objc-pointer-introspection-performSelector 430
-Wdeprecated-pragma 430
-Wdeprecated-register 430
-Wdeprecated-this-capture 431
-Wdeprecated-type 431
-Wdeprecated-volatile 431
-Wdeprecated-writable-strings 431
-Wdeprecated-xl-loop-pragmas 431
-Wdirect-ivar-access 431
-Wdisabled-macro-expansion 431
-Wdisabled-optimization 431
-Wdiscard-qual 432
-Wdistributed-object-modifiers 432
-Wdiv-by-zero 432
-Wdivision-by-zero 432
-Wdll-attribute-on-redeclaration 432
-Wdllexport-explicit-instantiation-decl 432
-Wdllimport-static-field-def 432
-Wdocumentation 432
-Wdocumentation-deprecated-sync 433
-Wdocumentation-html 433
-Wdocumentation-pedantic 433
-Wdocumentation-unknown-command 433
-Wdollar-in-identifier-extension 434
-Wdouble-promotion 434
-Wdtor-name 434
-Wdtor-typedef 434
-Wduplicate-decl-specifier 434
-Wduplicate-enum 434
-Wduplicate-method-arg 434
-Wduplicate-method-match 435
-Wduplicate-protocol 435
-Wdynamic-class-memaccess 435
-Wdynamic-exception-spec 435
-Weffc++ 435
-Welaborated-enum-base 435
-Welaborated-enum-class 435
-Wembedded-directive 435

-Wempty-body 435
-Wempty-decomposition 436
-Wempty-init-stmt 436
-Wempty-margins 436
-Wempty-translation-unit 436
-Wencode-type 436
-Wendif-labels 436
-Wenum-compare 436
-Wenum-compare-conditional 437
-Wenum-compare-switch 437
-Wenum-conversion 437
-Wenum-enum-conversion 437
-Wenum-float-conversion 437
-Wenum-too-large 437
-Wexceptions 437
-Wexcess-initializers 438
-Wexit-time-destructors 438
-Wexpansion-to-defined 438
-Wexplicit-initialize-call 438
-Wexplicit-ownership-type 438
-Wexport-unnamed 438
-Wexport-using-directive 438
-Wextern-c-compat 439
-Wextern-initializer 439
-Wextra 439
-Wextra-qualification 439
-Wextra-semi 439
-Wextra-semi-stmt 439
-Wextra-tokens 439
-Wfinal-dtor-non-final-class 440
-Wfinal-macro 440
-Wfixed-enum-extension 440
-Wfixed-point-overflow 440
-Wflag-enum 440
-Wflexible-array-extensions 440
-Wfloat-conversion 440
-Wfloat-equal 440
-Wfloat-overflow-conversion 441
-Wfloat-zero-conversion 441
-Wfor-loop-analysis 441
-Wformat 441
-Wformat-extra-args 442
-Wformat-insufficient-args 442
-Wformat-invalid-specifier 442
-Wformat-non-iso 442

-Wformat-nonliteral 442
-Wformat-pedantic 443
-Wformat-security 443
-Wformat-type-confusion 443
-Wformat-y2k 443
-Wformat-zero-length 443
-Wformat=2 443
-Wfortify-source 443
-Wfour-char-constants 443
-Wframe-address 444
-Wframe-larger-than 444
-Wframe-larger-than= 444
-Wframework-include-private-from-public 444
-Wfree-nonheap-object 444
-Wfunction-def-in-objc-container 444
-Wfunction-multiversion 444
-Wfuse-ld-path 444
-Wfuture-attribute-extensions 445
-Wfuture-compat 445
-Wgcc-compat 445
-Wglobal-constructors 445
-Wglobal-isel 445
-Wgnu 445
-Wgnu-alignof-expression 446
-Wgnu-anonymous-struct 446
-Wgnu-array-member-paren-init 446
-Wgnu-auto-type 446
-Wgnu-binary-literal 446
-Wgnu-case-range 446
-Wgnu-complex-integer 446
-Wgnu-compound-literal-initializer 446
-Wgnu-conditional-omitted-operand 447
-Wgnu-designator 447
-Wgnu-empty-initializer 447
-Wgnu-empty-struct 447
-Wgnu-flexible-array-initializer 447
-Wgnu-flexible-array-union-member 447
-Wgnu-folding-constant 447
-Wgnu-imaginary-constant 447
-Wgnu-include-next 448
-Wgnu-inline-cpp-without-extern 448
-Wgnu-label-as-value 448
-Wgnu-line-marker 448
-Wgnu-null-pointer-arithmetic 448
-Wgnu-pointer-arith 448

-Wgnu-redeclared-enum 448
-Wgnu-statement-expression 448
-Wgnu-statement-expression-from-macro-expansion 449
-Wgnu-static-float-init 449
-Wgnu-string-literal-operator-template 449
-Wgnu-union-cast 449
-Wgnu-variable-sized-type-not-at-end 449
-Wgnu-zero-line-directive 449
-Wgnu-zero-variadic-macro-arguments 449
-Wgpu-maybe-wrong-side 449
-Wheader-guard 449
-Wheader-hygiene 450
-Whip-only 450
-Whlsl-extensions 450
-Widiomatic-parentheses 450
-Wignored-attributes 450
-Wignored-availability-without-sdk-settings 453
-Wignored-optimization-argument 453
-Wignored-pragma-intrinsic 453
-Wignored-pragma-optimize 453
-Wignored-pragmas 453
-Wignored-qualifiers 455
-Wignored-reference-qualifiers 456
-Wimplicit 456
-Wimplicit-atomic-properties 456
-Wimplicit-const-int-float-conversion 456
-Wimplicit-conversion-floating-point-to-bool 456
-Wimplicit-exception-spec-mismatch 456
-Wimplicit-fallthrough 456
-Wimplicit-fallthrough-per-function 456
-Wimplicit-fixed-point-conversion 456
-Wimplicit-float-conversion 457
-Wimplicit-function-declaration 457
-Wimplicit-int 457
-Wimplicit-int-conversion 457
-Wimplicit-int-float-conversion 457
-Wimplicit-retain-self 458
-Wimplicitly-unsigned-literal 458
-Wimport 458
-Wimport-preprocessor-directive-pedantic 458
-Winaccessible-base 458
-Winclude-next-absolute-path 458
-Winclude-next-outside-header 458
-Wincompatible-exception-spec 458
-Wincompatible-function-pointer-types 458

-Wincompatible-library-redeclaration 459
-Wincompatible-ms-struct 459
-Wincompatible-pointer-types 459
-Wincompatible-pointer-types-discards-qualifiers 459
-Wincompatible-property-type 459
-Wincompatible-sysroot 459
-Wincomplete-framework-module-declaration 459
-Wincomplete-implementation 460
-Wincomplete-module 460
-Wincomplete-setjmp-declaration 460
-Wincomplete-umbrella 460
-Winconsistent-dllimport 460
-Winconsistent-missing-destructor-override 460
-Winconsistent-missing-override 460
-Wincrement-bool 460
-Winfinite-recursion 461
-Winit-self 461
-Winitializer-overrides 461
-Winjected-class-name 461
-Winline 461
-Winline-asm 461
-Winline-namespace-reopened-noninline 461
-Winline-new-delete 461
-Winstantiation-after-specialization 462
-Wint-conversion 462
-Wint-conversions 462
-Wint-in-bool-context 462
-Wint-to-pointer-cast 462
-Wint-to-void-pointer-cast 462
-Winteger-overflow 462
-Winterrupt-service-routine 462
-Winvalid-command-line-argument 463
-Winvalid-constexpr 463
-Winvalid-iboutlet 463
-Winvalid-initializer-from-system-header 463
-Winvalid-ios-deployment-target 463
-Winvalid-no-builtin-names 463
-Winvalid-noreturn 464
-Winvalid-offsetof 464
-Winvalid-or-nonexistent-directory 464
-Winvalid-partial-specialization 464
-Winvalid-pch 464
-Winvalid-pp-token 464
-Winvalid-source-encoding 464
-Winvalid-token-paste 465

-Wjump-seh-finally 465
-Wkeyword-compat 465
-Wkeyword-macro 465
-Wknr-promoted-parameter 465
-Wlanguage-extension-token 465
-Wlarge-by-value-copy 465
-Wliblto 465
-Wlinker-warnings 466
-Wliteral-conversion 466
-Wliteral-range 466
-Wlocal-type-template-args 466
-Wlogical-not-parentheses 466
-Wlogical-op-parentheses 466
-Wlong-long 466
-Wloop-analysis 467
-Wmacro-redefined 467
-Wmain 467
-Wmain-return-type 467
-Wmalformed-warning-check 467
-Wmany-braces-around-scalar-init 467
-Wmax-tokens 467
-Wmax-unsigned-zero 468
-Wmemset-transposed-args 468
-Wmemsize-comparison 468
-Wmethod-signatures 468
-Wmicrosoft 468
-Wmicrosoft-abstract 468
-Wmicrosoft-anon-tag 469
-Wmicrosoft-cast 469
-Wmicrosoft-charize 469
-Wmicrosoft-comment-paste 469
-Wmicrosoft-const-init 469
-Wmicrosoft-cpp-macro 469
-Wmicrosoft-default-arg-redefinition 469
-Wmicrosoft-drectve-section 469
-Wmicrosoft-end-of-file 470
-Wmicrosoft-enum-forward-reference 470
-Wmicrosoft-enum-value 470
-Wmicrosoft-exception-spec 470
-Wmicrosoft-exists 470
-Wmicrosoft-explicit-constructor-call 470
-Wmicrosoft-extra-qualification 470
-Wmicrosoft-fixed-enum 471
-Wmicrosoft-flexible-array 471
-Wmicrosoft-goto 471

-Wmicrosoft-inaccessible-base 471
-Wmicrosoft-include 471
-Wmicrosoft-mutable-reference 471
-Wmicrosoft-pure-definition 471
-Wmicrosoft-redeclare-static 471
-Wmicrosoft-sealed 471
-Wmicrosoft-static-assert 472
-Wmicrosoft-template 472
-Wmicrosoft-template-shadow 472
-Wmicrosoft-union-member-reference 472
-Wmicrosoft-unqualified-friend 472
-Wmicrosoft-using-decl 473
-Wmicrosoft-void-pseudo-dtor 473
-Wmisexpect 473
-Wmisleading-indentation 473
-Wmismatched-new-delete 473
-Wmismatched-parameter-types 473
-Wmismatched-return-types 473
-Wmismatched-tags 473
-Wmissing-braces 474
-Wmissing-constinit 474
-Wmissing-declarations 474
-Wmissing-exception-spec 474
-Wmissing-field-initializers 474
-Wmissing-format-attribute 474
-Wmissing-include-dirs 474
-Wmissing-method-return-type 474
-Wmissing-noescape 474
-Wmissing-noreturn 475
-Wmissing-prototype-for-cc 475
-Wmissing-prototypes 475
-Wmissing-selector-name 475
-Wmissing-sysroot 475
-Wmissing-variable-declarations 475
-Wmisspelled-assumption 475
-Rmodule-build 475
-Wmodule-conflict 476
-Wmodule-file-config-mismatch 476
-Wmodule-file-extension 476
-Rmodule-import 476
-Wmodule-import-in-extern-c 476
-Rmodule-lock 476
-Wmodules-ambiguous-internal-linkage 476
-Wmodules-import-nested-redundant 476
-Wmost 477

-Wmove 477
-Wmsvc-include 477
-Wmsvc-not-found 477
-Wmultichar 477
-Wmultiple-move-vbase 477
-Wnarrowing 477
-Wnested-anon-types 477
-Wnested-externs 477
-Wnew-returns-null 477
-Wnewline-eof 478
-Wnoderef 478
-Wnoexcept-type 478
-Wnon-c-typedef-for-linkage 478
-Wnon-gcc 478
-Wnon-literal-null-conversion 478
-Wnon-modular-include-in-framework-module 478
-Wnon-modular-include-in-module 478
-Wnon-pod-varargs 479
-Wnon-power-of-two-alignment 479
-Wnon-virtual-dtor 479
-Wnonnull 479
-Wnonportable-cfstrings 479
-Wnonportable-include-path 479
-Wnonportable-system-include-path 479
-Wnonportable-vector-initialization 479
-Wnontrivial-memaccess 480
-Wnsconsumed-mismatch 480
-Wnsreturns-mismatch 480
-Wnull-arithmetic 480
-Wnull-character 480
-Wnull-conversion 480
-Wnull-dereference 480
-Wnull-pointer-arithmetic 481
-Wnull-pointer-subtraction 481
-Wnullability 481
-Wnullability-completeness 481
-Wnullability-completeness-on-arrays 481
-Wnullability-declspec 481
-Wnullability-extension 481
-Wnullability-inferred-on-nested-type 482
-Wnullable-to-nonnull-conversion 482
-Wobjc-autosynthesis-property-ivar-name-match 482
-Wobjc-bool-constant-conversion 482
-Wobjc-boxing 482
-Wobjc-circular-container 482

-Wobjc-cocoa-api 482
-Wobjc-designated-initializers 482
-Wobjc-dictionary-duplicate-keys 483
-Wobjc-flexible-array 483
-Wobjc-forward-class-redefinition 483
-Wobjc-interface-ivars 483
-Wobjc-literal-compare 483
-Wobjc-literal-conversion 483
-Wobjc-macro-redefinition 483
-Wobjc-messaging-id 484
-Wobjc-method-access 484
-Wobjc-missing-property-synthesis 484
-Wobjc-missing-super-calls 484
-Wobjc-multiple-method-names 484
-Wobjc-noncopy-retain-block-property 484
-Wobjc-nonunified-exceptions 484
-Wobjc-property-assign-on-object-type 485
-Wobjc-property-implementation 485
-Wobjc-property-implicit-mismatch 485
-Wobjc-property-matches-cocoa-ownership-rule 485
-Wobjc-property-no-attribute 485
-Wobjc-property-synthesis 485
-Wobjc-protocol-method-implementation 485
-Wobjc-protocol-property-synthesis 486
-Wobjc-protocol-qualifiers 486
-Wobjc-readonly-with-setter-property 486
-Wobjc-redundant-api-use 486
-Wobjc-redundant-literal-use 486
-Wobjc-root-class 486
-Wobjc-signed-char-bool 486
-Wobjc-signed-char-bool-implicit-float-conversion 486
-Wobjc-signed-char-bool-implicit-int-conversion 487
-Wobjc-string-compare 487
-Wobjc-string-concatenation 487
-Wobjc-unsafe-perform-selector 487
-Wodr 487
-Wold-style-cast 488
-Wold-style-definition 488
-Wopencl-unsupported-rgba 488
-Wopenmp 488
-Wopenmp-51-extensions 488
-Wopenmp-clauses 488
-Wopenmp-loop-form 489
-Wopenmp-mapping 489
-Wopenmp-target 489

-Woption-ignored 489
-Wordered-compare-function-pointers 490
-Wout-of-line-declaration 490
-Wout-of-scope-function 490
-Wover-aligned 490
-Woverflow 491
-Woverlength-strings 491
-Woverloaded-shift-op-parentheses 491
-Woverloaded-virtual 491
-Woverride-init 491
-Woverride-module 491
-Woverriding-method-mismatch 491
-Woverriding-t-option 491
-Wpacked 492
-Wpadded 492
-Wparentheses 492
-Wparentheses-equality 492
-Wpartial-availability 492
-Rpass 492
-Rpass-analysis 492
-Wpass-failed 492
-Rpass-missed 493
-Wpch-date-time 493
-Wpedantic 493
-Wpedantic-core-features 496
-Wpedantic-macros 496
-Wpessimizing-move 496
-Wpointer-arith 496
-Wpointer-bool-conversion 496
-Wpointer-compare 496
-Wpointer-integer-compare 496
-Wpointer-sign 497
-Wpointer-to-enum-cast 497
-Wpointer-to-int-cast 497
-Wpointer-type-mismatch 497
-Wpoison-system-directories 497
-Wpotentially-direct-selector 497
-Wpotentially-evaluated-expression 497
-Wpragma-clang-attribute 497
-Wpragma-once-outside-header 498
-Wpragma-pack 498
-Wpragma-pack-suspicious-include 498
-Wpragma-system-header-outside-header 498
-Wpragmas 498
-Wpre-c++14-compat 498

-Wpre-c++14-compat-pedantic 499
-Wpre-c++17-compat 499
-Wpre-c++17-compat-pedantic 499
-Wpre-c++20-compat 500
-Wpre-c++20-compat-pedantic 500
-Wpre-c++2b-compat 501
-Wpre-c++2b-compat-pedantic 501
-Wpre-c2x-compat 501
-Wpre-c2x-compat-pedantic 501
-Wpre-openmp-51-compat 501
-Wpredefined-identifier-outside-function 501
-Wprivate-extern 502
-Wprivate-header 502
-Wprivate-module 502
-Wprofile-instr-missing 502
-Wprofile-instr-out-of-date 502
-Wprofile-instr-unprofiled 502
-Wproperty-access-dot-syntax 502
-Wproperty-attribute-mismatch 503
-Wprotocol 503
-Wprotocol-property-synthesis-ambiguity 503
-Wpsabi 503
-Wquoted-include-in-framework-header 503
-Wrange-loop-analysis 503
-Wrange-loop-bind-reference 503
-Wrange-loop-construct 503
-Wreadonly-iboutlet-property 504
-Wreceiver-expr 504
-Wreceiver-forward-class 504
-Wredeclared-class-member 504
-Wredundant-consteval-if 504
-Wredundant-decls 504
-Wredundant-move 504
-Wredundant-parens 504
-Wregister 504
-Wreinterpret-base-class 505
-Rremark-backend-plugin 505
-Wreorder 505
-Wreorder-ctor 505
-Wreorder-init-list 505
-Wrequires-super-attribute 505
-Wreserved-id-macro 505
-Wreserved-identifier 505
-Wreserved-macro-identifier 505
-Wreserved-user-defined-literal 506

-Wrestrict-expansion 506
-Wretained-language-linkage 506
-Wreturn-stack-address 506
-Wreturn-std-move 506
-Wreturn-type 506
-Wreturn-type-c-linkage 507
-Wrewrite-not-bool 507
-Rround-trip-cc1-args 507
-Wrtti 507
-Rsanitize-address 507
-Rsearch-path-usage 507
-Wsection 507
-Wselector 508
-Wselector-type-mismatch 508
-Wself-assign 508
-Wself-assign-field 508
-Wself-assign-overloaded 508
-Wself-move 508
-Wsemicolon-before-method-body 508
-Wsentinel 508
-Wsequence-point 509
-Wserialized-diagnostics 509
-Wshadow 509
-Wshadow-all 509
-Wshadow-field 509
-Wshadow-field-in-constructor 509
-Wshadow-field-in-constructor-modified 509
-Wshadow-ivar 509
-Wshadow-uncaptured-local 510
-Wshift-count-negative 510
-Wshift-count-overflow 510
-Wshift-negative-value 510
-Wshift-op-parentheses 510
-Wshift-overflow 510
-Wshift-sign-overflow 510
-Wshorten-64-to-32 510
-Wsign-compare 510
-Wsign-conversion 511
-Wsign-promo 511
-Wsigned-enum-bitfield 511
-Wsigned-unsigned-wchar 511
-Wsizeof-array-argument 511
-Wsizeof-array-decay 511
-Wsizeof-array-div 511
-Wsizeof-pointer-div 511

-Wsizeof-pointer-memaccess 512
-Wslash-u-filename 512
-Wslh-asm-goto 512
-Wsometimes-uninitialized 512
-Wsource-mgr 512
-Wsource-uses-openmp 512
-Wspir-compat 513
-Wspirv-compat 513
-Wstack-exhausted 513
-Wstack-protector 513
-Wstatic-float-init 513
-Wstatic-in-inline 513
-Wstatic-inline-explicit-instantiation 513
-Wstatic-local-in-inline 513
-Wstatic-self-init 514
-Wstdlibcxx-not-found 514
-Wstrict-aliasing 514
-Wstrict-aliasing=0 514
-Wstrict-aliasing=1 514
-Wstrict-aliasing=2 514
-Wstrict-overflow 514
-Wstrict-overflow=0 514
-Wstrict-overflow=1 514
-Wstrict-overflow=2 514
-Wstrict-overflow=3 514
-Wstrict-overflow=4 514
-Wstrict-overflow=5 514
-Wstrict-potentially-direct-selector 515
-Wstrict-prototypes 515
-Wstrict-selector-match 515
-Wstring-compare 515
-Wstring-concatenation 515
-Wstring-conversion 515
-Wstring-plus-char 515
-Wstring-plus-int 515
-Wstrlcpy-strlcat-size 516
-Wstrncat-size 516
-Wsuggest-destructor-override 516
-Wsuggest-override 516
-Wsuper-class-method-mismatch 516
-Wsuspicious-bzero 516
-Wsuspicious-memaccess 516
-Wswift-name-attribute 516
-Wswitch 517
-Wswitch-bool 517

-Wswitch-default 517
-Wswitch-enum 517
-Wsync-fetch-and-nand-semantics-changed 517
-Wsynth 518
-Wtarget-clones-mixed-specifiers 518
-Wtautological-bitwise-compare 518
-Wtautological-compare 518
-Wtautological-constant-compare 518
-Wtautological-constant-in-range-compare 518
-Wtautological-constant-out-of-range-compare 518
-Wtautological-objc-bool-compare 518
-Wtautological-overlap-compare 519
-Wtautological-pointer-compare 519
-Wtautological-type-limit-compare 519
-Wtautological-undefined-compare 519
-Wtautological-unsigned-char-zero-compare 519
-Wtautological-unsigned-enum-zero-compare 519
-Wtautological-unsigned-zero-compare 519
-Wtautological-value-range-compare 519
-Wtcb-enforcement 519
-Wtentative-definition-incomplete-type 520
-Wthread-safety 520
-Wthread-safety-analysis 520
-Wthread-safety-attributes 520
-Wthread-safety-beta 521
-Wthread-safety-negative 521
-Wthread-safety-precise 521
-Wthread-safety-reference 521
-Wthread-safety-verbose 521
-Wtrigraphs 521
-Wtype-limits 522
-Wtype-safety 522
-Wtypedef-redefinition 522
-Wtypename-missing 522
-Wunable-to-open-stats-file 522
-Wunaligned-access 522
-Wunaligned-qualifier-implicit-cast 522
-Wunavailable-declarations 522
-Wundeclared-selector 523
-Wundef 523
-Wundef-prefix 523
-Wundefined-bool-conversion 523
-Wundefined-func-template 523
-Wundefined-inline 523
-Wundefined-internal 523

-Wundefined-internal-type 523
-Wundefined-reinterpret-cast 523
-Wundefined-var-template 524
-Wunderaligned-exception-object 524
-Wunevaluated-expression 524
-Wunguarded-availability 524
-Wunguarded-availability-new 524
-Wunicode 524
-Wunicode-homoglyph 525
-Wunicode-whitespace 525
-Wunicode-zero-width 525
-Wuninitialized 525
-Wuninitialized-const-reference 525
-Wunknown-argument 525
-Wunknown-assumption 526
-Wunknown-attributes 526
-Wunknown-cuda-version 526
-Wunknown-directives 526
-Wunknown-escape-sequence 526
-Wunknown-pragmas 526
-Wunknown-sanitizers 527
-Wunknown-warning-option 527
-Wunnamed-type-template-args 527
-Wunneeded-internal-declaration 527
-Wunneeded-member-function 528
-Wunqualified-std-cast-call 528
-Wunreachable-code 528
-Wunreachable-code-aggressive 528
-Wunreachable-code-break 528
-Wunreachable-code-fallthrough 528
-Wunreachable-code-generic-assoc 528
-Wunreachable-code-loop-increment 528
-Wunreachable-code-return 528
-Wunsequenced 529
-Wunsupported-abi 529
-Wunsupported-abs 529
-Wunsupported-availability-guard 529
-Wunsupported-cb 529
-Wunsupported-dll-base-class-template 529
-Wunsupported-floating-point-opt 529
-Wunsupported-friend 530
-Wunsupported-gpopt 530
-Wunsupported-nan 530
-Wunsupported-target-opt 530
-Wunsupported-visibility 530

-Wunusable-partial-specialization 530
-Wunused 530
-Wunused-argument 531
-Wunused-but-set-parameter 531
-Wunused-but-set-variable 531
-Wunused-command-line-argument 531
-Wunused-comparison 531
-Wunused-const-variable 531
-Wunused-exception-parameter 532
-Wunused-function 532
-Wunused-getter-return-value 532
-Wunused-label 532
-Wunused-lambda-capture 532
-Wunused-local-typedef 532
-Wunused-local-typedefs 532
-Wunused-macros 532
-Wunused-member-function 532
-Wunused-parameter 532
-Wunused-private-field 533
-Wunused-property-ivar 533
-Wunused-result 533
-Wunused-template 533
-Wunused-value 533
-Wunused-variable 533
-Wunused-volatile-lvalue 534
-Wused-but-marked-unused 534
-Wuser-defined-literals 534
-Wuser-defined-warnings 534
-Wvarargs 534
-Wvariadic-macros 534
-Wvec-elem-size 534
-Wvector-conversion 535
-Wvector-conversions 535
-Wvexing-parse 535
-Wvisibility 535
-Wvla 535
-Wvla-extension 535
-Wvoid-pointer-to-enum-cast 535
-Wvoid-pointer-to-int-cast 535
-Wvoid-ptr-dereference 536
-Wvolatile-register-var 536
-Wwasm-exception-spec 536
-Wweak-template-vtables 536
-Wweak-vtables 536
-Wwritable-strings 536

-Wwrite-strings 536
-Wxor-used-as-pow 536
-Wzero-as-null-pointer-constant 536
-Wzero-length-array 537

Introduction

This page lists the diagnostic flags currently supported by Clang.

Diagnostic flags

-W

Synonym for -Wextra.

-W#pragma-messages

This diagnostic is enabled by default.

Diagnostic text:

The text of this diagnostic is not controlled by Clang.

-W#warnings

This diagnostic is enabled by default.

Diagnostic text:

The text of this diagnostic is not controlled by Clang.

-WCFString-literal

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-WCL4

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Wall, -Wextra.

-WIndependentClass-attribute

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-WNSObject-attribute

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wabi

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wabsolute-value

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table

-Wabstract-final-class

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wabstract-vbase-init

Diagnostic text:

Removed table

-Waddress

This diagnostic is enabled by default.

Controls -Wpointer-bool-conversion, -Wstring-compare, -Wtautological-pointer-compare.

-Waddress-of-packed-member

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Waddress-of-temporary

This diagnostic is an error by default, but the flag `-Wno-address-of-temporary` can be used to disable the error.

Diagnostic text:

Removed table

-Waggregate-return

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Waggressive-restrict

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Waix-compat

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Walign-mismatch

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wall

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Wmisleading-indentation, -Wmost, -Wparentheses, -Wswitch, -Wswitch-bool.

-Walloca

Diagnostic text:

Removed table

-Walloca-with-align-alignof

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Walways-inline-coroutine

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wambiguous-delete

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wambiguous-ellipsis

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wambiguous-macro

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wambiguous-member-template

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wambiguous-reversed-operator

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wanalyzer-incompatible-plugin

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wanon-enum-enum-conversion

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wdeprecated-anon-enum-enum-conversion.

Diagnostic text:

Removed table

-Wanonymous-pack-parens

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Warc

This diagnostic is enabled by default.

Controls -Warc-non-pod-memaccess, -Warc-retain-cycles, -Warc-unsafe-retained-assign.

-Warc-bridge-casts-disallowed-in-nonarc

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Warc-maybe-repeated-use-of-weak

Diagnostic text:

Removed table

-Warc-non-pod-memaccess

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Warc-performSelector-leaks

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Warc-repeated-use-of-weak

Also controls -Warc-maybe-repeated-use-of-weak.

Diagnostic text:

Removed table

-Warc-retain-cycles

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Warc-unsafe-retained-assign

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

-Wargument-outside-range

This diagnostic is an error by default, but the flag `-Wno-argument-outside-range` can be used to disable the error.

Diagnostic text:

Removed table

-Wargument-undefined-behaviour

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Warray-bounds

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

-Warray-bounds-pointer-arithmetic

Diagnostic text:

Removed table

Removed table

-Wasm

Synonym for -Wasm-operand-widths.

-Wasm-operand-widths

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wassign-enum

Diagnostic text:

Removed table

-Wassume

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wat-protocol

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Watimport-in-framework-header

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Watomic-access

This diagnostic is an error by default, but the flag `-Wno-atomic-access` can be used to disable the error.

Diagnostic text:

Removed table

-Watomic-alignment

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Watomic-implicit-seq-cst

Diagnostic text:

Removed table

-Watomic-memory-ordering

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Watomic-properties

Controls -Wcustom-atomic-properties, -Wimplicit-atomic-properties.

-Watomic-property-with-user-defined-accessor

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wattribute-packed-for-bitfield

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wattribute-warning

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wattributes

This diagnostic is enabled by default.

Controls -Wignored-attributes, -Wunknown-attributes.

-Wauto-disable-vptr-sanitizer

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wauto-import

Diagnostic text:

Removed table

-Wauto-storage-class

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wauto-var-id

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wavailability

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wavr-rtlib-linking-quirks

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wbackend-plugin

This diagnostic is enabled by default.

Diagnostic text:

The text of this diagnostic is not controlled by Clang.

-Wbackslash-newline-escape

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wbad-function-cast

Diagnostic text:

Removed table

-Wbinary-literal

Controls -Wc++14-binary-literal, -Wc++98-c++11-compat-binary-literal, -Wgnu-binary-literal.

-Wbind-to-temporary-copy

Also controls -Wc++98-compat-bind-to-temporary-copy.

Diagnostic text:

Removed table
Removed table

-Wbinding-in-condition

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wbit-int-extension

Diagnostic text:

Removed table

-Wbitfield-constant-conversion

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wbitfield-enum-conversion

Diagnostic text:

Removed table
Removed table
Removed table

-Wbitfield-width

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wbitwise-conditional-parentheses

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wbitwise-instead-of-logical

Diagnostic text:

Removed table

-Wbitwise-op-parentheses

Diagnostic text:

Removed table

-Wblock-capture-autoreleasing

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wbool-conversion

This diagnostic is enabled by default.

Also controls -Wpointer-bool-conversion, -Wundefined-bool-conversion.

Diagnostic text:

Removed table

-Wbool-conversions

Synonym for -Wbool-conversion.

-Wbool-operation

Also controls -Wbitwise-instead-of-logical.

Diagnostic text:

Removed table

-Wbraced-scalar-init

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wbranch-protection

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table

-Wbridge-cast

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table

-Wbuiltin-assume-aligned-alignment

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wbuiltin-macro-redefined

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wbuiltin-memcpy-chk-size

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wbuiltin-requires-header

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wc++-compat

Diagnostic text:

Removed table

-Wc++0x-compat

Synonym for -Wc++11-compat.

-Wc++0x-extensions

Synonym for -Wc++11-extensions.

-Wc++0x-narrowing

Synonym for -Wc++11-narrowing.

-Wc++11-compat

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wc++11-compat-deprecated-writable-strings, -Wc++11-compat-reserved-user-defined-literal, -Wc++11-narrowing, -Wpre-c++14-compat, -Wpre-c++17-compat, -Wpre-c++20-compat, -Wpre-c++2b-compat.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

-Wc++11-compat-deprecated-writable-strings

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wc++11-compat-pedantic

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Wc++11-compat, -Wpre-c++14-compat-pedantic, -Wpre-c++17-compat-pedantic, -Wpre-c++20-compat-pedantic, -Wpre-c++2b-compat-pedantic.

-Wc++11-compat-reserved-user-defined-literal

Diagnostic text:

Removed table

-Wc++11-extensions

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wc++11-extra-semi, -Wc++11-inline-namespace, -Wc++11-long-long.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wc++11-extra-semi

Diagnostic text:

Removed table

-Wc++11-inline-namespace

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wc++11-long-long

Diagnostic text:

Removed table

-Wc++11-narrowing

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wc++14-attribute-extensions

Diagnostic text:

Removed table

-Wc++14-binary-literal

Diagnostic text:

Removed table

-Wc++14-compat

Controls -Wpre-c++17-compat, -Wpre-c++20-compat, -Wpre-c++2b-compat.

-Wc++14-compat-pedantic

Controls -Wc++14-compat, -Wpre-c++17-compat-pedantic, -Wpre-c++20-compat-pedantic, -Wpre-c++2b-compat-pedantic.

-Wc++14-extensions

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wc++14-attribute-extensions, -Wc++14-binary-literal.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wc++17-attribute-extensions

Diagnostic text:

Removed table

-Wc++17-compat

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Wc++17-compat-mangling, -Wdeprecated-increment-bool, -Wdeprecated-register, -Wpre-c++20-compat, -Wpre-c++2b-compat.

-Wc++17-compat-mangling

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wc++17-compat-pedantic

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Wc++17-compat, -Wpre-c++20-compat-pedantic, -Wpre-c++2b-compat-pedantic.

-Wc++17-extensions

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wc++17-attribute-extensions.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wc++1y-extensions

Synonym for -Wc++14-extensions.

-Wc++1z-compat

Synonym for -Wc++17-compat.

-Wc++1z-compat-mangling

Synonym for -Wc++17-compat-mangling.

-Wc++1z-extensions

Synonym for -Wc++17-extensions.

-Wc++20-attribute-extensions

Diagnostic text:

Removed table

-Wc++20-compat

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wpre-c++2b-compat.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wc++20-compat-pedantic

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Wc++20-compat, -Wpre-c++2b-compat-pedantic.

-Wc++20-designator

Diagnostic text:

Removed table

-Wc++20-extensions

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wc++20-attribute-extensions, -Wc++20-designator.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

-Wc++2a-compat

Synonym for -Wc++20-compat.

-Wc++2a-compat-pedantic

Synonym for -Wc++20-compat-pedantic.

-Wc++2a-extensions

Synonym for -Wc++20-extensions.

-Wc++2b-extensions

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

-Wc++98-c++11-c++14-c++17-compat

Synonym for -Wpre-c++20-compat.

-Wc++98-c++11-c++14-c++17-compat-pedantic

Synonym for -Wpre-c++20-compat-pedantic.

-Wc++98-c++11-c++14-compat

Synonym for -Wpre-c++17-compat.

-Wc++98-c++11-c++14-compat-pedantic

Synonym for -Wpre-c++17-compat-pedantic.

-Wc++98-c++11-compat

Synonym for -Wpre-c++14-compat.

-Wc++98-c++11-compat-binary-literal

Diagnostic text:

Removed table

-Wc++98-c++11-compat-pedantic

Synonym for -Wpre-c++14-compat-pedantic.

-Wc++98-compat

Also controls -Wc++98-compat-local-type-template-args, -Wc++98-compat-unnamed-type-template-args, -Wpre-c++14-compat, -Wpre-c++17-compat, -Wpre-c++20-compat, -Wpre-c++2b-compat.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wc++98-compat-bind-to-temporary-copy

Diagnostic text:

Removed table

-Wc++98-compat-extra-semi

Diagnostic text:

Removed table

-Wc++98-compat-local-type-template-args

Diagnostic text:

Removed table

-Wc++98-compat-pedantic

Also controls -Wc++98-compat, -Wc++98-compat-bind-to-temporary-copy, -Wc++98-compat-extra-semi, -Wpre-c++14-compat-pedantic, -Wpre-c++17-compat-pedantic, -Wpre-c++20-compat-pedantic, -Wpre-c++2b-compat-pedantic.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wc++98-compat-unnamed-type-template-args

Diagnostic text:

Removed table

-Wc11-extensions

Diagnostic text:

Removed table
Removed table
Removed table
Removed table

-Wc2x-extensions

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table

-Wc99-compat

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table

-Wc99-designator

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wc++20-designator.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table

-Wc99-extensions

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wc99-designator.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wcall-to-pure-virtual-from-ctor-dtor

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wcalled-once-parameter

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wcompletion-handler.

Diagnostic text:

Removed table

Removed table

Removed table

-Wcast-align

Diagnostic text:

Removed table

-Wcast-calling-convention

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wcast-function-type

Diagnostic text:

Removed table

-Wcast-of-sel-type

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wcast-qual

Diagnostic text:

Removed table

Removed table

-Wcast-qual-unrelated

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wchar-align

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wchar-subscripts

Diagnostic text:

Removed table

Removed table

-Wclang-cl-pch

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table

-Wclass-conversion

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table

-Wclass-varargs

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wnon-pod-varargs.

Diagnostic text:

Removed table

-Wcmse-union-leak

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wcomma

Diagnostic text:

Removed table

-Wcomment

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table

-Wcomments

Synonym for -Wcomment.

-Wcompare-distinct-pointer-types

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wcompletion-handler

Diagnostic text:

Removed table
Removed table
Removed table

-Wcomplex-component-init

Diagnostic text:

Removed table

-Wcompound-token-split

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Wcompound-token-split-by-macro, -Wcompound-token-split-by-space.

-Wcompound-token-split-by-macro

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wcompound-token-split-by-space

Diagnostic text:

Removed table

-Wconcepts-ts-compat

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wconditional-type-mismatch

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wconditional-uninitialized

Diagnostic text:

Removed table

-Wconfig-macros

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wconstant-conversion

This diagnostic is enabled by default.

Also controls -Wbitfield-constant-conversion, -Wobjc-bool-constant-conversion.

Diagnostic text:

Removed table

-Wconstant-evaluated

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wconstant-logical-operand

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wconstexpr-not-const

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wconsumed

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

Removed table

Removed table

-Wconversion

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wbitfield-enum-conversion, -Wbool-conversion, -Wconstant-conversion, -Wenum-conversion, -Wfloat-conversion, -Wimplicit-float-conversion, -Wimplicit-int-conversion, -Wint-conversion, -Wliteral-conversion, -Wnon-literal-null-conversion, -Wnull-conversion, -Wobjc-literal-conversion, -Wshorten-64-to-32, -Wsign-conversion, -Wstring-conversion.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

-Wconversion-null

Synonym for -Wnull-conversion.

-Wcoroutine

This diagnostic is enabled by default.

Also controls -Walways-inline-coroutine, -Wcoroutine-missing-unhandled-exception, -Wdeprecated-coroutine.

Diagnostic text:

Removed table

-Wcoroutine-missing-unhandled-exception

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wcovered-switch-default

Diagnostic text:

Removed table

-Wcpp

Synonym for -W#warnings.

-Wcstring-format-directive

Diagnostic text:

Removed table

-Wctad-maybe-unsupported

Diagnostic text:

Removed table

-Wctor-dtor-privacy

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wctu

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wcuda-compat

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table

-Wcustom-atomic-properties

Diagnostic text:

Removed table

-Wcxx-attribute-extension

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdangling

This diagnostic is enabled by default.

Also controls -Wdangling-field, -Wdangling-gsl, -Wdangling-initializer-list, -Wreturn-stack-address.

Diagnostic text:

Removed table
Removed table

-Wdangling-else

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdangling-field

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table

-Wdangling-gsl

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table

-Wdangling-initializer-list

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdarwin-sdk-settings

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdate-time

Diagnostic text:

Removed table

-Wdealloc-in-category

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdebug-compression-unavailable

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdeclaration-after-statement

Diagnostic text:

Removed table

Removed table

-Wdefaulted-function-deleted

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wdelegating-ctor-cycles

This diagnostic is an error by default, but the flag `-Wno-delegating-ctor-cycles` can be used to disable the error.

Diagnostic text:

Removed table

-Wdelete-abstract-non-virtual-dtor

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdelete-incomplete

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wdelete-non-abstract-non-virtual-dtor

Diagnostic text:

Removed table

-Wdelete-non-virtual-dtor

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Wdelete-abstract-non-virtual-dtor, -Wdelete-non-abstract-non-virtual-dtor.

-Wdelimited-escape-sequence-extension

Diagnostic text:

Removed table

-Wdeprecate-lax-vec-conv-all

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdeprecated

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wdeprecated-anon-enum-enum-conversion, -Wdeprecated-array-compare, -Wdeprecated-attributes, -Wdeprecated-comma-subscript, -Wdeprecated-copy, -Wdeprecated-copy-with-dtor, -Wdeprecated-declarations, -Wdeprecated-dynamic-exception-spec, -Wdeprecated-enum-compare, -Wdeprecated-enum-compare-conditional, -Wdeprecated-enum-enum-conversion, -Wdeprecated-enum-float-conversion, -Wdeprecated-increment-bool, -Wdeprecated-pragma, -Wdeprecated-register, -Wdeprecated-this-capture, -Wdeprecated-type, -Wdeprecated-volatile, -Wdeprecated-writable-strings.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

-Wdeprecated-altivec-src-compat

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdeprecated-anon-enum-enum-conversion

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdeprecated-array-compare

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdeprecated-attributes

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wdeprecated-comma-subscript

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdeprecated-copy

Also controls -Wdeprecated-copy-with-user-provided-copy.

Diagnostic text:

Removed table

-Wdeprecated-copy-dtor

Synonym for -Wdeprecated-copy-with-dtor.

-Wdeprecated-copy-with-dtor

Also controls -Wdeprecated-copy-with-user-provided-dtor.

Diagnostic text:

Removed table

-Wdeprecated-copy-with-user-provided-copy

Diagnostic text:

Removed table

-Wdeprecated-copy-with-user-provided-dtor

Diagnostic text:

Removed table

-Wdeprecated-coroutine

This diagnostic is enabled by default.

Also controls -Wdeprecated-experimental-coroutine.

Diagnostic text:

Removed table

-Wdeprecated-declarations

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wdeprecated-dynamic-exception-spec

Diagnostic text:

Removed table

-Wdeprecated-enum-compare

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdeprecated-enum-compare-conditional

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdeprecated-enum-enum-conversion

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdeprecated-enum-float-conversion

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdeprecated-experimental-coroutine

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdeprecated-implementations

Diagnostic text:

Removed table

Removed table

-Wdeprecated-increment-bool

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdeprecated-non-prototype

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wdeprecated-objc-isa-usage

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wdeprecated-objc-pointer-introspection

This diagnostic is enabled by default.

Also controls -Wdeprecated-objc-pointer-introspection-performSelector.

Diagnostic text:

Removed table

-Wdeprecated-objc-pointer-introspection-performSelector

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdeprecated-pragma

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdeprecated-register

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdeprecated-this-capture

Diagnostic text:

Removed table

-Wdeprecated-type

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdeprecated-volatile

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

-Wdeprecated-writable-strings

Synonym for -Wc++11-compat-deprecated-writable-strings.

-Wdeprecated-xl-loop-pragmas

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdirect-ivar-access

Diagnostic text:

Removed table

-Wdisabled-macro-expansion

Diagnostic text:

Removed table

-Wdisabled-optimization

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wdiscard-qual

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wdistributed-object-modifiers

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wdiv-by-zero

Synonym for -Wdivision-by-zero.

-Wdivision-by-zero

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdll-attribute-on-redeclaration

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdllexport-explicit-instantiation-decl

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdllimport-static-field-def

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdocumentation

Also controls -Wdocumentation-deprecated-sync, -Wdocumentation-html.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

-Wdocumentation-deprecated-sync

Diagnostic text:

Removed table

-Wdocumentation-html

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

-Wdocumentation-pedantic

Also controls -Wdocumentation-unknown-command.

Diagnostic text:

Removed table

-Wdocumentation-unknown-command

Diagnostic text:

Removed table

Removed table

-Wdollar-in-identifier-extension

Diagnostic text:

Removed table

-Wdouble-promotion

Diagnostic text:

Removed table

-Wdtor-name

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

-Wdtor-typedef

This diagnostic is an error by default, but the flag `-Wno-dtor-typedef` can be used to disable the error.

Diagnostic text:

Removed table

-Wduplicate-decl-specifier

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

-Wduplicate-enum

Diagnostic text:

Removed table

-Wduplicate-method-arg

Diagnostic text:

Removed table

-Wduplicate-method-match

Diagnostic text:

Removed table

-Wduplicate-protocol

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdynamic-class-memaccess

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wdynamic-exception-spec

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wdeprecated-dynamic-exception-spec.

Diagnostic text:

Removed table

-Weffc++

Synonym for -Wnon-virtual-dtor.

-Welaborated-enum-base

This diagnostic is an error by default, but the flag `-Wno-elaborated-enum-base` can be used to disable the error.

Diagnostic text:

Removed table

-Welaborated-enum-class

This diagnostic is an error by default, but the flag `-Wno-elaborated-enum-class` can be used to disable the error.

Diagnostic text:

Removed table

-Wembedded-directive

Diagnostic text:

Removed table

-Wempty-body

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

-Wempty-decomposition

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wempty-init-stmt

Diagnostic text:

Removed table

-Wempty-margins

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wempty-translation-unit

Diagnostic text:

Removed table

-Wencode-type

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wendif-labels

Synonym for -Wextra-tokens.

-Wenum-compare

This diagnostic is enabled by default.

Also controls -Wdeprecated-enum-compare, -Wenum-compare-switch.

Diagnostic text:

Removed table

-Wenum-compare-conditional

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wdeprecated-enum-compare-conditional.

Diagnostic text:

Removed table

-Wenum-compare-switch

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wenum-conversion

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wenum-compare-conditional, -Wenum-enum-conversion, -Wenum-float-conversion.

Diagnostic text:

Removed table

-Wenum-enum-conversion

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wdeprecated-enum-enum-conversion.

Diagnostic text:

Removed table

-Wenum-float-conversion

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wdeprecated-enum-float-conversion.

Diagnostic text:

Removed table

-Wenum-too-large

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wexceptions

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

-Wexcess-initializers

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

-Wexit-time-destructors

Diagnostic text:

Removed table

-Wexpansion-to-defined

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table

Removed table

-Wexplicit-initialize-call

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wexplicit-ownership-type

Diagnostic text:

Removed table

-Wexport-unnamed

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wexport-using-directive

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wextern-c-compat

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wextern-initializer

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wextra

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wdeprecated-copy, -Wempty-init-stmt, -Wfuse-ld-path, -Wignored-qualifiers, -Winitializer-overrides, -Wmissing-field-initializers, -Wmissing-method-return-type, -Wnull-pointer-arithmetic, -Wnull-pointer-subtraction, -Wsemicolon-before-method-body, -Wsign-compare, -Wstring-concatenation, -Wunused-but-set-parameter, -Wunused-parameter.

Diagnostic text:

Removed table

-Wextra-qualification

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wextra-semi

Also controls -Wc++11-extra-semi, -Wc++98-compat-extra-semi.

Diagnostic text:

Removed table

Removed table

-Wextra-semi-stmt

Also controls -Wempty-init-stmt.

Diagnostic text:

Removed table

-Wextra-tokens

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wfinal-dtor-non-final-class

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wfinal-macro

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wfixed-enum-extension

Diagnostic text:

Removed table

-Wfixed-point-overflow

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wflag-enum

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wflexible-array-extensions

Diagnostic text:

Removed table

Removed table

-Wfloat-conversion

Also controls -Wfloat-overflow-conversion, -Wfloat-zero-conversion.

Diagnostic text:

Removed table

-Wfloat-equal

Diagnostic text:

Removed table

-Wfloat-overflow-conversion

Diagnostic text:

Removed table

Removed table

-Wfloat-zero-conversion

Diagnostic text:

Removed table

-Wfor-loop-analysis

Diagnostic text:

Removed table

Removed table

-Wformat

This diagnostic is enabled by default.

Also controls -Wformat-extra-args, -Wformat-insufficient-args, -Wformat-invalid-specifier, -Wformat-security, -Wformat-y2k, -Wformat-zero-length, -Wnonnull.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wformat-extra-args

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wformat-insufficient-args

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wformat-invalid-specifier

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wformat-non-iso

Diagnostic text:

Removed table
Removed table
Removed table

-Wformat-nonliteral

Diagnostic text:

Removed table

-Wformat-pedantic

Diagnostic text:

Removed table

Removed table

-Wformat-security

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wformat-type-confusion

Diagnostic text:

Removed table

-Wformat-y2k

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wformat-zero-length

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wformat=2

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Wformat-nonliteral, -Wformat-security, -Wformat-y2k.

-Wfortify-source

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

-Wfour-char-constants

Diagnostic text:

Removed table

-Wframe-address

Diagnostic text:

Removed table

-Wframe-larger-than

This diagnostic is enabled by default.

Diagnostic text:

The text of this diagnostic is not controlled by Clang.

Removed table

-Wframe-larger-than=

Synonym for -Wframe-larger-than.

-Wframework-include-private-from-public

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wfree-nonheap-object

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wfunction-def-in-objc-container

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wfunction-multiversion

This diagnostic is enabled by default.

Also controls -Wtarget-clones-mixed-specifiers.

Diagnostic text:

Removed table

Removed table

Removed table

-Wfuse-ld-path

Diagnostic text:

Removed table

-Wfuture-attribute-extensions

Controls -Wc++14-attribute-extensions, -Wc++17-attribute-extensions, -Wc++20-attribute-extensions.

-Wfuture-compat

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wgcc-compat

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

-Wglobal-constructors

Diagnostic text:

Removed table

Removed table

-Wglobal-isel

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wgnu

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Wgnu-alignof-expression, -Wgnu-anonymous-struct, -Wgnu-auto-type, -Wgnu-binary-literal, -Wgnu-case-range, -Wgnu-complex-integer, -Wgnu-compound-literal-initializer, -Wgnu-conditional-omitted-operand, -Wgnu-designator, -Wgnu-empty-initializer, -Wgnu-empty-struct, -Wgnu-flexible-array-initializer,

-Wgnu-flexible-array-union-member, -Wgnu-folding-constant, -Wgnu-imaginary-constant, -Wgnu-include-next, -Wgnu-label-as-value, -Wgnu-line-marker, -Wgnu-null-pointer-arithmetic, -Wgnu-pointer-arith, -Wgnu-redeclared-enum, -Wgnu-statement-expression, -Wgnu-static-float-init, -Wgnu-string-literal-operator-template, -Wgnu-union-cast, -Wgnu-variable-sized-type-not-at-end, -Wgnu-zero-line-directive, -Wgnu-zero-variadic-macro-arguments, -Wredeclared-class-member, -Wvla-extension, -Wzero-length-array.

-Wgnu-alignof-expression

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wgnu-anonymous-struct

Diagnostic text:

Removed table

-Wgnu-array-member-paren-init

This diagnostic is an error by default, but the flag `-Wno-gnu-array-member-paren-init` can be used to disable the error.

Diagnostic text:

Removed table

-Wgnu-auto-type

Diagnostic text:

Removed table

-Wgnu-binary-literal

Diagnostic text:

Removed table

-Wgnu-case-range

Diagnostic text:

Removed table

-Wgnu-complex-integer

Diagnostic text:

Removed table

-Wgnu-compound-literal-initializer

Diagnostic text:

Removed table

-Wgnu-conditional-omitted-operand

Diagnostic text:

Removed table

-Wgnu-designator

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table

-Wgnu-empty-initializer

Diagnostic text:

Removed table

-Wgnu-empty-struct

Diagnostic text:

Removed table
Removed table
Removed table

-Wgnu-flexible-array-initializer

Diagnostic text:

Removed table

-Wgnu-flexible-array-union-member

Diagnostic text:

Removed table

-Wgnu-folding-constant

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table

-Wgnu-imaginary-constant

Diagnostic text:

Removed table

-Wgnu-include-next

Diagnostic text:

Removed table

-Wgnu-inline-cpp-without-extern

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wgnu-label-as-value

Diagnostic text:

Removed table

Removed table

-Wgnu-line-marker

Diagnostic text:

Removed table

-Wgnu-null-pointer-arithmetic

Diagnostic text:

Removed table

-Wgnu-pointer-arith

Diagnostic text:

Removed table

Removed table

Removed table

-Wgnu-redeclared-enum

Diagnostic text:

Removed table

-Wgnu-statement-expression

Also controls -Wgnu-statement-expression-from-macro-expansion.

Diagnostic text:

Removed table

-Wgnu-statement-expression-from-macro-expansion

Diagnostic text:

Removed table

-Wgnu-static-float-init

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wgnu-string-literal-operator-template

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wgnu-union-cast

Diagnostic text:

Removed table

-Wgnu-variable-sized-type-not-at-end

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wgnu-zero-line-directive

Diagnostic text:

Removed table

-Wgnu-zero-variadic-macro-arguments

Diagnostic text:

Removed table

Removed table

-Wgpu-maybe-wrong-side

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wheader-guard

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wheader-hygiene

Diagnostic text:

Removed table

-Whip-only

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Whlsl-extensions

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Widiomatic-parentheses

Diagnostic text:

Removed table

-Wignored-attributes

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

-Wignored-availability-without-sdk-settings

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wignored-optimization-argument

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wignored-pragma-intrinsic

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wignored-pragma-optimize

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wignored-pragmas

This diagnostic is enabled by default.

Also controls -Wignored-pragma-intrinsic, -Wignored-pragma-optimize.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

-Wignored-qualifiers

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wignored-reference-qualifiers.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

-Wignored-reference-qualifiers

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wimplicit

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Wimplicit-function-declaration, -Wimplicit-int.

-Wimplicit-atomic-properties

Diagnostic text:

Removed table

Removed table

-Wimplicit-const-int-float-conversion

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wimplicit-conversion-floating-point-to-bool

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wimplicit-exception-spec-mismatch

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wimplicit-fallthrough

Also controls -Wimplicit-fallthrough-per-function.

Diagnostic text:

Removed table

-Wimplicit-fallthrough-per-function

Diagnostic text:

Removed table

-Wimplicit-fixed-point-conversion

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wimplicit-float-conversion

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wimplicit-int-float-conversion, -Wobjc-signed-char-bool-implicit-float-conversion.

Diagnostic text:

Removed table
Removed table

-Wimplicit-function-declaration

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table

-Wimplicit-int

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table

-Wimplicit-int-conversion

Also controls -Wobjc-signed-char-bool-implicit-int-conversion.

Diagnostic text:

Removed table
Removed table

-Wimplicit-int-float-conversion

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wimplicit-const-int-float-conversion.

Diagnostic text:

Removed table

-Wimplicit-retain-self

Diagnostic text:

Removed table

-Wimplicitly-unsigned-literal

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wimport

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wimport-preprocessor-directive-pedantic

Diagnostic text:

Removed table

-Winaccessible-base

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Winclude-next-absolute-path

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Winclude-next-outside-header

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wincompatible-exception-spec

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wincompatible-function-pointer-types

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wincompatible-library-redeclaration

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wincompatible-ms-struct

This diagnostic is an error by default, but the flag `-Wno-incompatible-ms-struct` can be used to disable the error.

Diagnostic text:

Removed table

Removed table

-Wincompatible-pointer-types

This diagnostic is enabled by default.

Also controls -Wincompatible-function-pointer-types, -Wincompatible-pointer-types-discards-qualifiers.

Diagnostic text:

Removed table

-Wincompatible-pointer-types-discards-qualifiers

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

-Wincompatible-property-type

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wincompatible-sysroot

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wincomplete-framework-module-declaration

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wincomplete-implementation

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wincomplete-module

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Wincomplete-umbrella, -Wnon-modular-include-in-module.

-Wincomplete-setjmp-declaration

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wincomplete-umbrella

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

-Winconsistent-dllimport

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Winconsistent-missing-destructor-override

Diagnostic text:

Removed table

-Winconsistent-missing-override

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wincrement-bool

This diagnostic is enabled by default.

Also controls -Wdeprecated-increment-bool.

Diagnostic text:

Removed table

-Winfinite-recursion

Diagnostic text:

Removed table

-Winit-self

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Winitializer-overrides

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Winjected-class-name

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Winline

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Winline-asm

This diagnostic is enabled by default.

Diagnostic text:

The text of this diagnostic is not controlled by Clang.

-Winline-namespace-reopened-noninline

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Winline-new-delete

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Winstantiation-after-specialization

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wint-conversion

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wint-conversions

Synonym for -Wint-conversion.

-Wint-in-bool-context

Diagnostic text:

Removed table

Removed table

-Wint-to-pointer-cast

This diagnostic is enabled by default.

Also controls -Wint-to-void-pointer-cast.

Diagnostic text:

Removed table

-Wint-to-void-pointer-cast

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Winteger-overflow

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Winterrupt-service-routine

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Winvalid-command-line-argument

This diagnostic is enabled by default.

Also controls -Wignored-optimization-argument.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Winvalid-constexpr

This diagnostic is an error by default, but the flag `-Wno-invalid-constexpr` can be used to disable the error.

Diagnostic text:

Removed table

-Winvalid-iboutlet

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table

-Winvalid-initializer-from-system-header

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Winvalid-ios-deployment-target

This diagnostic is an error by default, but the flag `-Wno-invalid-ios-deployment-target` can be used to disable the error.

Diagnostic text:

Removed table

-Winvalid-no-builtin-names

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Winvalid-noreturn

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Winvalid-offsetof

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Winvalid-or-nonexistent-directory

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table

Removed table

-Winvalid-partial-specialization

This diagnostic is an error by default, but the flag `-Wno-invalid-partial-specialization` can be used to disable the error.

Diagnostic text:

Removed table

-Winvalid-pch

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Winvalid-pp-token

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Winvalid-source-encoding

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Winvalid-token-paste

This diagnostic is an error by default, but the flag `-Wno-invalid-token-paste` can be used to disable the error.

Diagnostic text:

Removed table

-Wjump-seh-finally

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wkeyword-compat

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wkeyword-macro

Diagnostic text:

Removed table

-Wknr-promoted-parameter

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wlanguage-extension-token

Diagnostic text:

Removed table

-Wlarge-by-value-copy

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wliblto

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wlinker-warnings

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wliteral-conversion

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wliteral-range

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

-Wlocal-type-template-args

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wc++98-compat-local-type-template-args.

Diagnostic text:

Removed table

-Wlogical-not-parentheses

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wlogical-op-parentheses

Diagnostic text:

Removed table

-Wlong-long

Also controls -Wc++11-long-long.

Diagnostic text:

Removed table

-Wloop-analysis

Controls -Wfor-loop-analysis, -Wrange-loop-analysis.

-Wmacro-redefined

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmain

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wmain-return-type

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmalformed-warning-check

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmany-braces-around-scalar-init

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmax-tokens

Diagnostic text:

Removed table
Removed table

The warning is issued if the number of pre-processor tokens exceeds the token limit, which can be set in three ways:

1. As a limit at a specific point in a file, using the `clang max_tokens_here` pragma:
2. As a per-translation unit limit, using the `-fmax-tokens=` command-line flag:
3. As a per-translation unit limit using the `clang max_tokens_total` pragma, which works like and overrides the `-fmax-tokens=` flag:

These limits can be helpful in limiting code growth through included files.

Setting a token limit of zero means no limit.

Note that the warning is disabled by default, so -Wmax-tokens must be used in addition with the pragmas or -fmax-tokens flag to get any warnings.

-Wmax-unsigned-zero

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmemset-transposed-args

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmemsize-comparison

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmethod-signatures

Diagnostic text:

Removed table

Removed table

-Wmicrosoft

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Winconsistent-dllimport, -Wmicrosoft-abstract, -Wmicrosoft-anon-tag, -Wmicrosoft-cast, -Wmicrosoft-charize, -Wmicrosoft-comment-paste, -Wmicrosoft-const-init, -Wmicrosoft-cpp-macro, -Wmicrosoft-default-arg-redefinition, -Wmicrosoft-directve-section, -Wmicrosoft-end-of-file, -Wmicrosoft-enum-forward-reference, -Wmicrosoft-enum-value, -Wmicrosoft-exception-spec, -Wmicrosoft-explicit-constructor-call, -Wmicrosoft-extra-qualification, -Wmicrosoft-fixed-enum, -Wmicrosoft-flexible-array, -Wmicrosoft-goto, -Wmicrosoft-include, -Wmicrosoft-mutable-reference, -Wmicrosoft-pure-definition, -Wmicrosoft-redeclare-static, -Wmicrosoft-sealed, -Wmicrosoft-static-assert, -Wmicrosoft-template, -Wmicrosoft-union-member-reference, -Wmicrosoft-unqualified-friend, -Wmicrosoft-using-decl, -Wmicrosoft-void-pseudo-dtor.

-Wmicrosoft-abstract

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmicrosoft-anon-tag

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table
Removed table

-Wmicrosoft-cast

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table

-Wmicrosoft-charize

Diagnostic text:

Removed table

-Wmicrosoft-comment-paste

Diagnostic text:

Removed table

-Wmicrosoft-const-init

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmicrosoft-cpp-macro

Diagnostic text:

Removed table

-Wmicrosoft-default-arg-redefinition

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmicrosoft-drectve-section

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmicrosoft-end-of-file

Diagnostic text:

Removed table

-Wmicrosoft-enum-forward-reference

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmicrosoft-enum-value

Diagnostic text:

Removed table

-Wmicrosoft-exception-spec

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

-Wmicrosoft-exists

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmicrosoft-explicit-constructor-call

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmicrosoft-extra-qualification

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmicrosoft-fixed-enum

Diagnostic text:

Removed table

-Wmicrosoft-flexible-array

Diagnostic text:

Removed table

Removed table

-Wmicrosoft-goto

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmicrosoft-inaccessible-base

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmicrosoft-include

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmicrosoft-mutable-reference

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmicrosoft-pure-definition

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmicrosoft-redeclare-static

Diagnostic text:

Removed table

-Wmicrosoft-sealed

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmicrosoft-static-assert

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmicrosoft-template

This diagnostic is enabled by default.

Also controls -Wmicrosoft-template-shadow.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wmicrosoft-template-shadow

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmicrosoft-union-member-reference

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmicrosoft-unqualified-friend

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmicrosoft-using-decl

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmicrosoft-void-pseudo-dtor

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmisexpect

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmisleading-indentation

Diagnostic text:

Removed table

-Wmismatched-new-delete

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmismatched-parameter-types

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmismatched-return-types

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmismatched-tags

Diagnostic text:

Removed table

Removed table

-Wmissing-braces

Diagnostic text:

Removed table

-Wmissing-constinit

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmissing-declarations

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

-Wmissing-exception-spec

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmissing-field-initializers

Diagnostic text:

Removed table

-Wmissing-format-attribute

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wmissing-include-dirs

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wmissing-method-return-type

Diagnostic text:

Removed table

-Wmissing-noescape

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmissing-noreturn

Diagnostic text:

Removed table

Removed table

-Wmissing-prototype-for-cc

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmissing-prototypes

Diagnostic text:

Removed table

-Wmissing-selector-name

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmissing-sysroot

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmissing-variable-declarations

Diagnostic text:

Removed table

-Wmisspelled-assumption

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Rmodule-build

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

-Wmodule-conflict

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wmodule-file-config-mismatch

This diagnostic is an error by default, but the flag `-Wno-module-file-config-mismatch` can be used to disable the error.

Diagnostic text:

Removed table

-Wmodule-file-extension

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Rmodule-import

Diagnostic text:

Removed table

-Wmodule-import-in-extern-c

This diagnostic is an error by default, but the flag `-Wno-module-import-in-extern-c` can be used to disable the error.

Diagnostic text:

Removed table

-Rmodule-lock

Diagnostic text:

Removed table

-Wmodules-ambiguous-internal-linkage

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmodules-import-nested-redundant

This diagnostic is an error by default, but the flag `-Wno-modules-import-nested-redundant` can be used to disable the error.

Diagnostic text:

Removed table

-Wmost

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Wbool-operation, -Wcast-of-sel-type, -Wchar-subscripts, -Wcomment, -Wdelete-non-virtual-dtor, -Wextern-c-compat, -Wfor-loop-analysis, -Wformat, -Wframe-address, -Wimplicit, -Winfinite-recursion, -Wint-in-bool-context, -Wmismatched-tags, -Wmissing-braces, -Wmove, -Wmultichar, -Wobjc-designated-initializers, -Wobjc-flexible-array, -Wobjc-missing-super-calls, -Woverloaded-virtual, -Wprivate-extern, -Wrange-loop-construct, -Wreorder, -Wreturn-type, -Wself-assign, -Wself-move, -Wsizeof-array-argument, -Wsizeof-array-decay, -Wstring-plus-int, -Wtautological-compare, -Wtrigraphs, -Wuninitialized, -Wunknown-pragmas, -Wunused, -Wuser-defined-warnings, -Wvolatile-register-var.

-Wmove

Controls -Wpessimizing-move, -Wredundant-move, -Wreturn-std-move, -Wself-move.

-Wmsvc-include

Synonym for -Wmicrosoft-include.

-Wmsvc-not-found

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmultichar

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wmultiple-move-vbase

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wnarrowing

Synonym for -Wc++11-narrowing.

-Wnested-anon-types

Diagnostic text:

Removed table

-Wnested-externs

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wnew-returns-null

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wnewline-eof

Diagnostic text:

Removed table

Removed table

-Wnoderef

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

-Wnoexcept-type

Synonym for -Wc++17-compat-mangling.

-Wnon-c-typedef-for-linkage

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wnon-gcc

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Wconversion, -Wliteral-range, -Wsign-compare.

-Wnon-literal-null-conversion

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wnon-modular-include-in-framework-module

Diagnostic text:

Removed table

-Wnon-modular-include-in-module

Also controls -Wnon-modular-include-in-framework-module.

Diagnostic text:

Removed table

-Wnon-pod-varargs

This diagnostic is an error by default, but the flag `-Wno-non-pod-varargs` can be used to disable the error.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

-Wnon-power-of-two-alignment

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wnon-virtual-dtor

Diagnostic text:

Removed table

-Wnonnull

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wnonportable-cfstrings

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wnonportable-include-path

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wnonportable-system-include-path

Diagnostic text:

Removed table

-Wnonportable-vector-initialization

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wnontrivial-memaccess

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wnsconsumed-mismatch

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wnsreturns-mismatch

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wnull-arithmetic

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wnull-character

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wnull-conversion

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wnull-dereference

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wnull-pointer-arithmetic

Also controls -Wgnu-null-pointer-arithmetic.

Diagnostic text:

Removed table

-Wnull-pointer-subtraction

Diagnostic text:

Removed table

-Wnullability

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table

-Wnullability-completeness

This diagnostic is enabled by default.

Also controls -Wnullability-completeness-on-arrays.

Diagnostic text:

Removed table

-Wnullability-completeness-on-arrays

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wnullability-declspec

This diagnostic is an error by default, but the flag `-Wno-nullability-declspec` can be used to disable the error.

Diagnostic text:

Removed table

-Wnullability-extension

Diagnostic text:

Removed table

-Wnullability-inferred-on-nested-type

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wnullable-to-nonnull-conversion

Diagnostic text:

Removed table

-Wobjc-autosynthesis-property-ivar-name-match

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-bool-constant-conversion

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-boxing

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-circular-container

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-cocoa-api

Synonym for -Wobjc-redundant-api-use.

-Wobjc-designated-initializers

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table

Removed table

Removed table

-Wobjc-dictionary-duplicate-keys

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-flexible-array

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wobjc-forward-class-redefinition

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-interface-ivars

Diagnostic text:

Removed table

-Wobjc-literal-compare

This diagnostic is enabled by default.

Also controls -Wobjc-string-compare.

Diagnostic text:

Removed table

-Wobjc-literal-conversion

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wobjc-macro-redefinition

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-messaging-id

Diagnostic text:

Removed table

-Wobjc-method-access

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wobjc-missing-property-synthesis

Diagnostic text:

Removed table

-Wobjc-missing-super-calls

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-multiple-method-names

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-noncopy-retain-block-property

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-nonunified-exceptions

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-property-assign-on-object-type

Diagnostic text:

Removed table

-Wobjc-property-implementation

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table

-Wobjc-property-implicit-mismatch

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-property-matches-cocoa-ownership-rule

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-property-no-attribute

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table

-Wobjc-property-synthesis

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table

-Wobjc-protocol-method-implementation

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-protocol-property-synthesis

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-protocol-qualifiers

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-readonly-with-setter-property

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-redundant-api-use

Synonym for -Wobjc-redundant-literal-use.

-Wobjc-redundant-literal-use

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-root-class

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-signed-char-bool

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Wobjc-bool-constant-conversion, -Wobjc-signed-char-bool-implicit-float-conversion, -Wobjc-signed-char-bool-implicit-int-conversion, -Wtautological-objc-bool-compare.

-Wobjc-signed-char-bool-implicit-float-conversion

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-signed-char-bool-implicit-int-conversion

Diagnostic text:

Removed table

-Wobjc-string-compare

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-string-concatenation

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wobjc-unsafe-perform-selector

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wodr

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

Removed table
Removed table
Removed table
Removed table

-Wold-style-cast

Diagnostic text:

Removed table

-Wold-style-definition

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wopencl-unsupported-rgba

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wopenmp

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Wopenmp-51-extensions, -Wopenmp-clauses, -Wopenmp-loop-form, -Wopenmp-mapping, -Wopenmp-target, -Wsource-uses-openmp.

-Wopenmp-51-extensions

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wopenmp-clauses

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

-Wopenmp-loop-form

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wopenmp-mapping

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wopenmp-target

This diagnostic is enabled by default.

Also controls -Wopenmp-mapping.

Diagnostic text:

Removed table

Removed table

Removed table

-Woption-ignored

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wordered-compare-function-pointers

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table

-Wout-of-line-declaration

This diagnostic is an error by default, but the flag `-Wno-out-of-line-declaration` can be used to disable the error.

Diagnostic text:

Removed table

-Wout-of-scope-function

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wover-aligned

Diagnostic text:

Removed table

-Woverflow

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Woverlength-strings

Diagnostic text:

Removed table

-Woverloaded-shift-op-parentheses

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Woverloaded-virtual

Diagnostic text:

Removed table

-Woverride-init

Synonym for -Winitializer-overrides.

-Woverride-module

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Woverriding-method-mismatch

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Woverriding-t-option

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wpacked

Diagnostic text:

Removed table

-Wpadded

Diagnostic text:

Removed table

Removed table

Removed table

-Wparentheses

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wbitwise-conditional-parentheses, -Wbitwise-op-parentheses, -Wdangling-else, -Wlogical-not-parentheses, -Wlogical-op-parentheses, -Woverloaded-shift-op-parentheses, -Wparentheses-equality, -Wshift-op-parentheses.

Diagnostic text:

Removed table

Removed table

Removed table

-Wparentheses-equality

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wpartial-availability

Synonym for -Wunguarded-availability.

-Rpass

Diagnostic text:

The text of this diagnostic is not controlled by Clang.

-Rpass-analysis

Diagnostic text:

The text of this diagnostic is not controlled by Clang.

Removed table

Removed table

-Wpass-failed

This diagnostic is enabled by default.

Diagnostic text:

The text of this diagnostic is not controlled by Clang.

-Rpass-missed

Diagnostic text:

The text of this diagnostic is not controlled by Clang.

-Wpch-date-time

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wpedantic

Also controls -Wbit-int-extension, -Wc++11-extra-semi, -Wc++11-long-long, -Wc++14-attribute-extensions, -Wc++14-binary-literal, -Wc++17-attribute-extensions, -Wc++20-attribute-extensions, -Wc++20-designator, -Wc11-extensions, -Wcomplex-component-init, -Wdelimited-escape-sequence-extension, -Wdollar-in-identifier-extension, -Wembedded-directive, -Wempty-translation-unit, -Wfixed-enum-extension, -Wflexible-array-extensions, -Wfuture-attribute-extensions, -Wgnu-anonymous-struct, -Wgnu-auto-type, -Wgnu-binary-literal, -Wgnu-case-range, -Wgnu-complex-integer, -Wgnu-compound-literal-initializer, -Wgnu-conditional-omitted-operand, -Wgnu-empty-initializer, -Wgnu-empty-struct, -Wgnu-flexible-array-initializer, -Wgnu-flexible-array-union-member, -Wgnu-imaginary-constant, -Wgnu-include-next, -Wgnu-label-as-value, -Wgnu-line-marker, -Wgnu-null-pointer-arithmetic, -Wgnu-pointer-arith, -Wgnu-redeclared-enum, -Wgnu-statement-expression, -Wgnu-union-cast, -Wgnu-zero-line-directive, -Wgnu-zero-variadic-macro-arguments, -Wimport-preprocessor-directive-pedantic, -Wkeyword-macro, -Wlanguage-extension-token, -Wlong-long, -Wmicrosoft-charize, -Wmicrosoft-comment-paste, -Wmicrosoft-cpp-macro, -Wmicrosoft-end-of-file, -Wmicrosoft-enum-value, -Wmicrosoft-fixed-enum, -Wmicrosoft-flexible-array, -Wmicrosoft-redeclare-static, -Wnested-anon-types, -Wnullability-extension, -Woverlength-strings, -Wretained-language-linkage, -Wundefined-internal-type, -Wvla-extension, -Wzero-length-array.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

Removed table
Removed table
Removed table
Removed table

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wpedantic-core-features

Diagnostic text:

Removed table

Removed table

-Wpedantic-macros

This diagnostic is enabled by default.

Controls -Wbuiltin-macro-redefined, -Wdeprecated-pragma, -Wfinal-macro, -Wmacro-redefined, -Wrestrict-expansion.

-Wpessimizing-move

Diagnostic text:

Removed table

Removed table

-Wpointer-arith

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wgnu-pointer-arith.

Diagnostic text:

Removed table

Removed table

Removed table

-Wpointer-bool-conversion

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wpointer-compare

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wpointer-integer-compare

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wpointer-sign

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wpointer-to-enum-cast

This diagnostic is enabled by default.

Also controls -Wvoid-pointer-to-enum-cast.

Diagnostic text:

Removed table

-Wpointer-to-int-cast

This diagnostic is enabled by default.

Also controls -Wpointer-to-enum-cast, -Wvoid-pointer-to-int-cast.

Diagnostic text:

Removed table

-Wpointer-type-mismatch

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wpoison-system-directories

Diagnostic text:

Removed table

-Wpotentially-direct-selector

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wpotentially-evaluated-expression

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wpragma-clang-attribute

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wpragma-once-outside-header

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wpragma-pack

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wpragma-pack-suspicious-include.

Diagnostic text:

Removed table

Removed table

-Wpragma-pack-suspicious-include

Diagnostic text:

Removed table

-Wpragma-system-header-outside-header

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wpragmas

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wignored-pragmas, -Wpragma-clang-attribute, -Wpragma-pack, -Wunknown-pragmas.

Diagnostic text:

Removed table

Removed table

-Wpre-c++14-compat

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

-Wpre-c++14-compat-pedantic

Controls -Wc++98-c++11-compat-binary-literal, -Wpre-c++14-compat.

-Wpre-c++17-compat

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

-Wpre-c++17-compat-pedantic

Also controls -Wpre-c++17-compat.

Diagnostic text:

Removed table

Removed table

-Wpre-c++20-compat

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wpre-c++20-compat-pedantic

Also controls -Wpre-c++20-compat.

Diagnostic text:

Removed table
Removed table

-Wpre-c++2b-compat

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wpre-c++2b-compat-pedantic

Synonym for -Wpre-c++2b-compat.

-Wpre-c2x-compat

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table

-Wpre-c2x-compat-pedantic

Synonym for -Wpre-c2x-compat.

-Wpre-openmp-51-compat

Diagnostic text:

Removed table

-Wpredefined-identifier-outside-function

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wprivate-extern

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wprivate-header

This diagnostic is an error by default, but the flag `-Wno-private-header` can be used to disable the error.

Diagnostic text:

Removed table

-Wprivate-module

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

-Wprofile-instr-missing

Diagnostic text:

Removed table

-Wprofile-instr-out-of-date

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wprofile-instr-unprofiled

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wproperty-access-dot-syntax

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wproperty-attribute-mismatch

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table

-Wprotocol

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wprotocol-property-synthesis-ambiguity

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wpsabi

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wquoted-include-in-framework-header

Diagnostic text:

Removed table

-Wrange-loop-analysis

Controls -Wrange-loop-bind-reference, -Wrange-loop-construct.

-Wrange-loop-bind-reference

Diagnostic text:

Removed table

-Wrange-loop-construct

Diagnostic text:

Removed table
Removed table

-Wreadonly-iboutlet-property

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wreceiver-expr

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wreceiver-forward-class

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table

Removed table

-Wredeclared-class-member

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wredundant-consteval-if

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wredundant-decls

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wredundant-move

Diagnostic text:

Removed table

-Wredundant-parens

Diagnostic text:

Removed table

-Wregister

This diagnostic is enabled by default.

Also controls -Wdeprecated-register.

Diagnostic text:

Removed table

-Wreinterpret-base-class

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Rremark-backend-plugin

Diagnostic text:

The text of this diagnostic is not controlled by Clang.

-Wreorder

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Wreorder-ctor, -Wreorder-init-list.

-Wreorder-ctor

Diagnostic text:

Removed table

Removed table

-Wreorder-init-list

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wrequires-super-attribute

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wreserved-id-macro

Synonym for -Wreserved-macro-identifier.

-Wreserved-identifier

Also controls -Wreserved-macro-identifier.

Diagnostic text:

Removed table

-Wreserved-macro-identifier

Diagnostic text:

Removed table

-Wreserved-user-defined-literal

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wc++11-compat-reserved-user-defined-literal.

Diagnostic text:

Removed table

Removed table

-Wrestrict-expansion

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wretained-language-linkage

Diagnostic text:

Removed table

-Wreturn-stack-address

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

-Wreturn-std-move

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wreturn-type

This diagnostic is enabled by default.

Also controls -Wreturn-type-c-linkage.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

-Wreturn-type-c-linkage

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wrewrite-not-bool

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Rround-trip-cc1-args

Diagnostic text:

Removed table

-Wrtti

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Rsanitize-address

Diagnostic text:

Removed table

Removed table

-Rsearch-path-usage

Diagnostic text:

Removed table

-Wsection

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

-Wselector

Also controls -Wselector-type-mismatch.

Diagnostic text:

Removed table

-Wselector-type-mismatch

Diagnostic text:

Removed table

-Wself-assign

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wself-assign-field, -Wself-assign-overloaded.

Diagnostic text:

Removed table

-Wself-assign-field

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wself-assign-overloaded

Diagnostic text:

Removed table

-Wself-move

Diagnostic text:

Removed table

-Wsemicolon-before-method-body

Diagnostic text:

Removed table

-Wsentinel

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wsequence-point

Synonym for -Wunsequenced.

-Wserialized-diagnostics

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

-Wshadow

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wshadow-field-in-constructor-modified, -Wshadow-ivar.

Diagnostic text:

Removed table

-Wshadow-all

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Wshadow, -Wshadow-field, -Wshadow-field-in-constructor, -Wshadow-uncaptured-local.

-Wshadow-field

Diagnostic text:

Removed table

-Wshadow-field-in-constructor

Also controls -Wshadow-field-in-constructor-modified.

Diagnostic text:

Removed table

-Wshadow-field-in-constructor-modified

Diagnostic text:

Removed table

-Wshadow-ivar

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wshadow-uncaptured-local

Diagnostic text:

Removed table

-Wshift-count-negative

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wshift-count-overflow

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wshift-negative-value

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wshift-op-parentheses

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wshift-overflow

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wshift-sign-overflow

Diagnostic text:

Removed table

-Wshorten-64-to-32

Diagnostic text:

Removed table

-Wsign-compare

Diagnostic text:

Removed table

-Wsign-conversion

Diagnostic text:

Removed table

Removed table

Removed table

-Wsign-promo

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wsigned-enum-bitfield

Diagnostic text:

Removed table

-Wsigned-unsigned-wchar

This diagnostic is an error by default, but the flag `-Wno-signed-unsigned-wchar` can be used to disable the error.

Diagnostic text:

Removed table

-Wsizeof-array-argument

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wsizeof-array-decay

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wsizeof-array-div

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wsizeof-pointer-div

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wsizeof-pointer-memaccess

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wslash-u-filename

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wslh-asm-goto

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wsometimes-uninitialized

Diagnostic text:

Removed table

-Wsource-mgr

This diagnostic is enabled by default.

Diagnostic text:

The text of this diagnostic is not controlled by Clang.

-Wsource-uses-openmp

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

-Wspir-compat

Diagnostic text:

Removed table

-Wspirv-compat

Synonym for -Wspir-compat.

-Wstack-exhausted

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wstack-protector

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wstatic-float-init

This diagnostic is enabled by default.

Also controls -Wgnu-static-float-init.

Diagnostic text:

Removed table

-Wstatic-in-inline

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table

Removed table

-Wstatic-inline-explicit-instantiation

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wstatic-local-in-inline

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wstatic-self-init

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wstdlibcxx-not-found

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wstrict-aliasing

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wstrict-aliasing=0

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wstrict-aliasing=1

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wstrict-aliasing=2

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wstrict-overflow

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wstrict-overflow=0

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wstrict-overflow=1

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wstrict-overflow=2

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wstrict-overflow=3

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wstrict-overflow=4

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wstrict-overflow=5

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wstrict-potentially-direct-selector

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wpotentially-direct-selector.

Diagnostic text:

Removed table

-Wstrict-prototypes

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wdeprecated-non-prototype.

Diagnostic text:

Removed table

-Wstrict-selector-match

Diagnostic text:

Removed table

-Wstring-compare

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wstring-concatenation

Diagnostic text:

Removed table

-Wstring-conversion

Diagnostic text:

Removed table

-Wstring-plus-char

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wstring-plus-int

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wstrlcpy-strlcat-size

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wstrncat-size

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

-Wsuggest-destructor-override

Diagnostic text:

Removed table

-Wsuggest-override

Diagnostic text:

Removed table

-Wsuper-class-method-mismatch

Diagnostic text:

Removed table

-Wsuspicious-bzero

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wsuspicious-memaccess

This diagnostic is enabled by default.

Controls -Wdynamic-class-memaccess, -Wmemset-transposed-args, -Wnontrivial-memaccess, -Wsizeof-pointer-memaccess, -Wsuspicious-bzero.

-Wswift-name-attribute

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wswitch

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table

-Wswitch-bool

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wswitch-default

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wswitch-enum

Diagnostic text:

Removed table

-Wsync-fetch-and-nand-semantics-changed

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wsynth

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wtarget-clones-mixed-specifiers

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wtautological-bitwise-compare

Diagnostic text:

Removed table
Removed table

-Wtautological-compare

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wtautological-bitwise-compare, -Wtautological-constant-compare, -Wtautological-objc-bool-compare, -Wtautological-overlap-compare, -Wtautological-pointer-compare, -Wtautological-undefined-compare.

Diagnostic text:

Removed table
Removed table

-Wtautological-constant-compare

This diagnostic is enabled by default.

Also controls -Wtautological-constant-out-of-range-compare.

Diagnostic text:

Removed table
Removed table
Removed table

-Wtautological-constant-in-range-compare

Controls -Wtautological-value-range-compare, -Wtype-limits.

-Wtautological-constant-out-of-range-compare

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wtautological-objc-bool-compare

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wtautological-overlap-compare

Diagnostic text:

Removed table

-Wtautological-pointer-compare

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wtautological-type-limit-compare

Diagnostic text:

Removed table

-Wtautological-undefined-compare

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wtautological-unsigned-char-zero-compare

Diagnostic text:

Removed table

-Wtautological-unsigned-enum-zero-compare

Diagnostic text:

Removed table

-Wtautological-unsigned-zero-compare

Diagnostic text:

Removed table

-Wtautological-value-range-compare

Diagnostic text:

Removed table

-Wtcb-enforcement

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wtentative-definition-incomplete-type

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wthread-safety

Controls -Wthread-safety-analysis, -Wthread-safety-attributes, -Wthread-safety-precise, -Wthread-safety-reference.

-Wthread-safety-analysis

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wthread-safety-attributes

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

-Wthread-safety-beta

Diagnostic text:

Removed table

-Wthread-safety-negative

Diagnostic text:

Removed table

-Wthread-safety-precise

Diagnostic text:

Removed table

Removed table

Removed table

-Wthread-safety-reference

Diagnostic text:

Removed table

Removed table

-Wthread-safety-verbose

Diagnostic text:

Removed table

-Wtrigraphs

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

-Wtype-limits

Controls -Wtautological-type-limit-compare, -Wtautological-unsigned-char-zero-compare, -Wtautological-unsigned-enum-zero-compare, -Wtautological-unsigned-zero-compare.

-Wtype-safety

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

-Wtypedef-redefinition

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wtypename-missing

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunable-to-open-stats-file

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunaligned-access

Diagnostic text:

Removed table

-Wunaligned-qualifier-implicit-cast

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunavailable-declarations

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wundeclared-selector

Diagnostic text:

Removed table

Removed table

-Wundef

Diagnostic text:

Removed table

-Wundef-prefix

Diagnostic text:

Removed table

-Wundefined-bool-conversion

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wundefined-func-template

Diagnostic text:

Removed table

-Wundefined-inline

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wundefined-internal

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wundefined-internal-type

Diagnostic text:

Removed table

-Wundefined-reinterpret-cast

Diagnostic text:

Removed table

Removed table

-Wundefined-var-template

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunderaligned-exception-object

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunevaluated-expression

This diagnostic is enabled by default.

Also controls -Wpotentially-evaluated-expression.

Diagnostic text:

Removed table

-Wunguarded-availability

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wunguarded-availability-new.

Diagnostic text:

Removed table

-Wunguarded-availability-new

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunicode

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

-Wunicode-homoglyph

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunicode-whitespace

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunicode-zero-width

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wuninitialized

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wsometimes-uninitialized, -Wstatic-self-init, -Wuninitialized-const-reference.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

-Wuninitialized-const-reference

Diagnostic text:

Removed table

-Wunknown-argument

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wunknown-assumption

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunknown-attributes

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunknown-cuda-version

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wunknown-directives

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunknown-escape-sequence

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunknown-pragmas

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wunknown-sanitizers

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunknown-warning-option

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table

-Wunnamed-type-template-args

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wc++98-compat-unnamed-type-template-args.

Diagnostic text:

Removed table

-Wunneeded-internal-declaration

Diagnostic text:

Removed table

Removed table

-Wunneeded-member-function

Diagnostic text:

Removed table

-Wunqualified-std-cast-call

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunreachable-code

Some of the diagnostics controlled by this flag are enabled by default.

Also controls -Wunreachable-code-fallthrough, -Wunreachable-code-generic-assoc, -Wunreachable-code-loop-increment.

Diagnostic text:

Removed table

-Wunreachable-code-aggressive

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Wunreachable-code, -Wunreachable-code-break, -Wunreachable-code-return.

-Wunreachable-code-break

Diagnostic text:

Removed table

-Wunreachable-code-fallthrough

Diagnostic text:

Removed table

-Wunreachable-code-generic-assoc

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunreachable-code-loop-increment

Diagnostic text:

Removed table

-Wunreachable-code-return

Diagnostic text:

Removed table

-Wunsequenced

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wunsupported-abi

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunsupported-abs

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wunsupported-availability-guard

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunsupported-cb

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunsupported-dll-base-class-template

Diagnostic text:

Removed table

-Wunsupported-floating-point-opt

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wunsupported-friend

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wunsupported-gpopt

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunsupported-nan

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wunsupported-target-opt

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wunsupported-visibility

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunusable-partial-specialization

This diagnostic is an error by default, but the flag `-Wno-unusable-partial-specialization` can be used to disable the error.

Diagnostic text:

Removed table

-Wunused

Some of the diagnostics controlled by this flag are enabled by default.

Controls -Wunused-argument, -Wunused-but-set-variable, -Wunused-function, -Wunused-label, -Wunused-lambda-capture, -Wunused-local-typedef, -Wunused-private-field, -Wunused-property-ivar, -Wunused-value, -Wunused-variable.

-Wunused-argument

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wunused-but-set-parameter

Diagnostic text:

Removed table

-Wunused-but-set-variable

Diagnostic text:

Removed table

-Wunused-command-line-argument

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table
Removed table

-Wunused-comparison

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunused-const-variable

Diagnostic text:

Removed table

-Wunused-exception-parameter

Diagnostic text:

Removed table

-Wunused-function

Also controls -Wunneeded-internal-declaration.

Diagnostic text:

Removed table

-Wunused-getter-return-value

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wunused-label

Diagnostic text:

Removed table

-Wunused-lambda-capture

Diagnostic text:

Removed table

-Wunused-local-typedef

Diagnostic text:

Removed table

-Wunused-local-typedefs

Synonym for -Wunused-local-typedef.

-Wunused-macros

Diagnostic text:

Removed table

-Wunused-member-function

Also controls -Wunneeded-member-function.

Diagnostic text:

Removed table

-Wunused-parameter

Diagnostic text:

Removed table

-Wunused-private-field

Diagnostic text:

Removed table

-Wunused-property-ivar

Diagnostic text:

Removed table

-Wunused-result

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wunused-template

Also controls -Wunneeded-internal-declaration.

Diagnostic text:

Removed table

-Wunused-value

This diagnostic is enabled by default.

Also controls -Wunevaluated-expression, -Wunused-comparison, -Wunused-result.

Diagnostic text:

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

Removed table

-Wunused-variable

Also controls -Wunused-const-variable.

Diagnostic text:

Removed table

-Wunused-volatile-lvalue

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wused-but-marked-unused

Diagnostic text:

Removed table

-Wuser-defined-literals

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wuser-defined-warnings

This diagnostic is enabled by default.

Diagnostic text:

The text of this diagnostic is not controlled by Clang.

-Wvarargs

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

-Wvariadic-macros

Some of the diagnostics controlled by this flag are enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

-Wvec-elem-size

This diagnostic is an error by default, but the flag `-Wno-vec-elem-size` can be used to disable the error.

Diagnostic text:

Removed table

-Wvector-conversion

Diagnostic text:

Removed table

-Wvector-conversions

Synonym for -Wvector-conversion.

-Wvexing-parse

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

Removed table

-Wvisibility

This diagnostic is enabled by default.

Diagnostic text:

Removed table

Removed table

-Wvla

Also controls -Wvla-extension.

Diagnostic text:

Removed table

-Wvla-extension

Diagnostic text:

Removed table

-Wvoid-pointer-to-enum-cast

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wvoid-pointer-to-int-cast

This diagnostic is enabled by default.

Also controls -Wvoid-pointer-to-enum-cast.

Diagnostic text:

Removed table

-Wvoid-ptr-dereference

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wvolatile-register-var

This diagnostic flag exists for GCC compatibility, and has no effect in Clang.

-Wwasm-exception-spec

This diagnostic is enabled by default.

Diagnostic text:

Removed table

-Wweak-template-vtables

Diagnostic text:

Removed table

-Wweak-vtables

Diagnostic text:

Removed table

-Wwritable-strings

This diagnostic is enabled by default.

Also controls -Wdeprecated-writable-strings.

Diagnostic text:

Removed table

-Wwrite-strings

Synonym for -Wwritable-strings.

-Wxor-used-as-pow

This diagnostic is enabled by default.

Diagnostic text:

Removed table
Removed table
Removed table

-Wzero-as-null-pointer-constant

Diagnostic text:

Removed table

-Wzero-length-array

Diagnostic text:

Removed table

Cross-compilation using Clang

Introduction

This document will guide you in choosing the right Clang options for cross-compiling your code to a different architecture. It assumes you already know how to compile the code in question for the host architecture, and that you know how to choose additional include and library paths.

However, this document is *not* a “how to” and won’t help you setting your build system or Makefiles, nor choosing the right CMake options, etc. Also, it does not cover all the possible options, nor does it contain specific examples for specific architectures. For a concrete example, the instructions for cross-compiling LLVM itself may be of interest.

After reading this document, you should be familiar with the main issues related to cross-compilation, and what main compiler options Clang provides for performing cross-compilation.

Cross compilation issues

In GCC world, every host/target combination has its own set of binaries, headers, libraries, etc. So, it’s usually simple to download a package with all files in, unzip to a directory and point the build system to that compiler, that will know about its location and find all it needs to when compiling your code.

On the other hand, Clang/LLVM is natively a cross-compiler, meaning that one set of programs can compile to all targets by setting the `-target` option. That makes it a lot easier for programmers wishing to compile to different platforms and architectures, and for compiler developers that only have to maintain one build system, and for OS distributions, that need only one set of main packages.

But, as is true to any cross-compiler, and given the complexity of different architectures, OS’s and options, it’s not always easy finding the headers, libraries or binutils to generate target specific code. So you’ll need special options to help Clang understand what target you’re compiling to, where your tools are, etc.

Another problem is that compilers come with standard libraries only (like `compiler-rt`, `libcxx`, `libgcc`, `libm`, etc), so you’ll have to find and make available to the build system, every other library required to build your software, that is specific to your target. It’s not enough to have your host’s libraries installed.

Finally, not all toolchains are the same, and consequently, not every Clang option will work magically. Some options, like `--sysroot` (which effectively changes the logical root for headers and libraries), assume all your binaries and libraries are in the same directory, which may not true when your cross-compiler was installed by the distribution’s package management. So, for each specific case, you may use more than one option, and in most cases, you’ll end up setting include paths (`-I`) and library paths (`-L`) manually.

To sum up, different toolchains can:

- be host/target specific or more flexible
- be in a single directory, or spread out across your system
- have different sets of libraries and headers by default
- need special options, which your build system won’t be able to figure out by itself

General Cross-Compilation Options in Clang

Target Triple

The basic option is to define the target architecture. For that, use `-target <triple>`. If you don’t specify the target, CPU names won’t match (since Clang assumes the host triple), and the compilation will go ahead, creating code for the host platform, which will break later on when assembling or linking.

The triple has the general format `<arch><sub>-<vendor>-<sys>-<abi>`**, where:**

- `arch` = `x86_64`, `i386`, `arm`, `thumb`, `mips`, etc.
- `sub` = for ex. on ARM: `v5`, `v6m`, `v7a`, `v7m`, etc.
- `vendor` = `pc`, `apple`, `nvidia`, `ibm`, etc.
- `sys` = `none`, `linux`, `win32`, `darwin`, `cuda`, etc.
- `abi` = `eabi`, `gnu`, `android`, `macho`, `elf`, etc.

The sub-architecture options are available for their own architectures, of course, so “x86v7a” doesn’t make sense. The vendor needs to be specified only if there’s a relevant change, for instance between PC and Apple. Most of the time it can be omitted (and Unknown) will be assumed, which sets the defaults for the specified architecture. The system name is generally the OS (linux, darwin), but could be special like the bare-metal “none”.

When a parameter is not important, it can be omitted, or you can choose `unknown` and the defaults will be used. If you choose a parameter that Clang doesn’t know, like `blerg`, it’ll ignore and assume `unknown`, which is not always desired, so be careful.

Finally, the ABI option is something that will pick default CPU/FPU, define the specific behaviour of your code (PCS, extensions), and also choose the correct library calls, etc.

CPU, FPU, ABI

Once your target is specified, it’s time to pick the hardware you’ll be compiling to. For every architecture, a default set of CPU/FPU/ABI will be chosen, so you’ll almost always have to change it via flags.

Typical flags include:

- `-mcpu=<cpu-name>`, like x86-64, swift, cortex-a15
- `-mfpu=<fpu-name>`, like SSE3, NEON, controlling the FP unit available
- `-mfloat-abi=<fabi>`, like soft, hard, controlling which registers to use for floating-point

The default is normally the common denominator, so that Clang doesn’t generate code that breaks. But that also means you won’t get the best code for your specific hardware, which may mean orders of magnitude slower than you expect.

For example, if your target is `arm-none-eabi`, the default CPU will be `arm7tdmi` using soft float, which is extremely slow on modern cores, whereas if your triple is `armv7a-none-eabi`, it’ll be Cortex-A8 with NEON, but still using soft-float, which is much better, but still not great.

Toolchain Options

There are three main options to control access to your cross-compiler: `--sysroot`, `-I`, and `-L`. The two last ones are well known, but they’re particularly important for additional libraries and headers that are specific to your target.

There are two main ways to have a cross-compiler:

1. When you have extracted your cross-compiler from a zip file into a directory, you have to use `--sysroot=<path>`. The path is the root directory where you have unpacked your file, and Clang will look for the directories `bin`, `lib`, `include` in there.

   In this case, your setup should be pretty much done (if no additional headers or libraries are needed), as Clang will find all binaries it needs (assembler, linker, etc) in there.

2. When you have installed via a package manager (modern Linux distributions have cross-compiler packages available), make sure the target triple you set is *also* the prefix of your cross-compiler toolchain.

   In this case, Clang will find the other binaries (assembler, linker), but not always where the target headers and libraries are. People add system-specific clues to Clang often, but as things change, it’s more likely that it won’t find than the other way around.

   So, here, you’ll be a lot safer if you specify the include/library directories manually (via `-I` and `-L`).

Target-Specific Libraries

All libraries that you compile as part of your build will be cross-compiled to your target, and your build system will probably find them in the right place. But all dependencies that are normally checked against (like `libxml` or `libz` etc) will match against the host platform, not the target.

So, if the build system is not aware that you want to cross-compile your code, it will get every dependency wrong, and your compilation will fail during build time, not configure time.

Also, finding the libraries for your target are not as easy as for your host machine. There aren't many cross-libraries available as packages to most OS's, so you'll have to either cross-compile them from source, or download the package for your target platform, extract the libraries and headers, put them in specific directories and add `-I` and `-L` pointing to them.

Also, some libraries have different dependencies on different targets, so configuration tools to find dependencies in the host can get the list wrong for the target platform. This means that the configuration of your build can get things wrong when setting their own library paths, and you'll have to augment it via additional flags (configure, Make, CMake, etc).

Multilibs

When you want to cross-compile to more than one configuration, for example hard-float-ARM and soft-float-ARM, you'll have to have multiple copies of your libraries and (possibly) headers.

Some Linux distributions have support for Multilib, which handle that for you in an easier way, but if you're not careful and, for instance, forget to specify `-ccc-gcc-name armv7l-linux-gnueabihf-gcc` (which uses hard-float), Clang will pick the `armv7l-linux-gnueabi-ld` (which uses soft-float) and linker errors will happen.

The same is true if you're compiling for different ABIs, like `gnueabi` and `androideabi`, and might even link and run, but produce run-time errors, which are much harder to track down and fix.

Clang Static Analyzer

The Clang Static Analyzer is a source code analysis tool that finds bugs in C, C++, and Objective-C programs. It implements *path-sensitive*, *inter-procedural analysis* based on *symbolic execution* technique.

This is the Static Analyzer documentation page.

See the Official Tool Page.

Available Checkers

The analyzer performs checks that are categorized into families or "checkers".

The default set of checkers covers a variety of checks targeted at finding security and API usage bugs, dead code, and other logic errors. See the Default Checkers checkers list below.

In addition to these, the analyzer contains a number of Experimental Checkers (aka *alpha* checkers). These checkers are under development and are switched off by default. They may crash or emit a higher number of false positives.

The debug package contains checkers for analyzer developers for debugging purposes.

Table of Contents

Available Checkers **539**
Default Checkers 544
core 544
core.CallAndMessage (C, C++, ObjC) 544
core.DivideZero (C, C++, ObjC) 545
core.NonNullParamChecker (C, C++, ObjC) 545
core.NullDereference (C, C++, ObjC) 545
core.StackAddressEscape (C) 546
core.UndefinedBinaryOperatorResult (C) 546
core.VLASize (C) 547
core.uninitialized.ArraySubscript (C) 547
core.uninitialized.Assign (C) 547
core.uninitialized.Branch (C) 547
core.uninitialized.CapturedBlockVariable (C) 547
core.uninitialized.UndefReturn (C) 547
cplusplus 548
cplusplus.InnerPointer (C++) 548
cplusplus.NewDelete (C++) 548
cplusplus.NewDeleteLeaks (C++) 549
cplusplus.PlacementNewChecker (C++) 549
cplusplus.SelfAssignment (C++) 549
cplusplus.StringChecker (C++) 549
deadcode 549
deadcode.DeadStores (C) 549
nullability 550
nullability.NullPassedToNonnull (ObjC) 550
nullability.NullReturnedFromNonnull (ObjC) 550
nullability.NullableDereferenced (ObjC) 550
nullability.NullablePassedToNonnull (ObjC) 550
nullability.NullableReturnedFromNonnull (ObjC) 551
optin 551
optin.cplusplus.UninitializedObject (C++) 551
optin.cplusplus.VirtualCall (C++) 552
optin.mpi.MPI-Checker (C) 553
optin.osx.cocoa.localizability.EmptyLocalizationContextChecker (ObjC) 553
optin.osx.cocoa.localizability.NonLocalizedStringChecker (ObjC) 553
optin.performance.GCDAntipattern 553
optin.performance.Padding 554
optin.portability.UnixAPI 554
security 554
security.FloatLoopCounter (C) 554
security.insecureAPI.UncheckedReturn (C) 554
security.insecureAPI.bcmp (C) 554

security.insecureAPI.bcopy (C) 554
security.insecureAPI.bzero (C) 554
security.insecureAPI.getpw (C) 554
security.insecureAPI.gets (C) 554
security.insecureAPI.mkstemp (C) 555
security.insecureAPI.mktemp (C) 555
security.insecureAPI.rand (C) 555
security.insecureAPI.strcpy (C) 555
security.insecureAPI.vfork (C) 555
security.insecureAPI.DeprecatedOrUnsafeBufferHandling (C) 555
unix 555
unix.API (C) 556
unix.Malloc (C) 556
unix.MallocSizeof (C) 557
unix.MismatchedDeallocator (C, C++) 557
unix.Vfork (C) 558
unix.cstring.BadSizeArg (C) 558
unix.cstring.NullArg (C) 559
osx 559
osx.API (C) 559
osx.NumberObjectConversion (C, C++, ObjC) 559
osx.ObjCProperty (ObjC) 559
osx.SecKeychainAPI (C) 559
osx.cocoa.AtSync (ObjC) 560
osx.cocoa.AutoreleaseWrite 561
osx.cocoa.ClassRelease (ObjC) 561
osx.cocoa.Dealloc (ObjC) 561
osx.cocoa.IncompatibleMethodTypes (ObjC) 562
osx.cocoa.Loops 562
osx.cocoa.MissingSuperCall (ObjC) 562
osx.cocoa.NSAutoreleasePool (ObjC) 562
osx.cocoa.NSError (ObjC) 562
osx.cocoa.NilArg (ObjC) 563
osx.cocoa.NonNilReturnValue 563
osx.cocoa.ObjCGenerics (ObjC) 563
osx.cocoa.RetainCount (ObjC) 563
osx.cocoa.RunLoopAutoreleaseLeak 564
osx.cocoa.SelfInit (ObjC) 564
osx.cocoa.SuperDealloc (ObjC) 564
osx.cocoa.UnusedIvars (ObjC) 564
osx.cocoa.VariadicMethodTypes (ObjC) 565
osx.coreFoundation.CFError (C) 565
osx.coreFoundation.CFNumber (C) 565
osx.coreFoundation.CFRetainRelease (C) 565
osx.coreFoundation.containers.OutOfBounds (C) 565

osx.coreFoundation.containers.PointerSizedValues (C) 565
Fuchsia 566
fuchsia.HandleChecker 566
WebKit 566
webkit.RefCntblBaseVirtualDtor 566
webkit.NoUncountedMemberChecker 566
webkit.UncountedLambdaCapturesChecker 567
Experimental Checkers 567
alpha.clone 567
alpha.clone.CloneChecker (C, C++, ObjC) 567
alpha.core 567
alpha.core.BoolAssignment (ObjC) 567
alpha.core.C11Lock 568
alpha.core.CallAndMessageUnInitRefArg (C,C++, ObjC) 568
alpha.core.CastSize (C) 568
alpha.core.CastToStruct (C, C++) 568
alpha.core.Conversion (C, C++, ObjC) 568
alpha.core.DynamicTypeChecker (ObjC) 569
alpha.core.FixedAddr (C) 569
alpha.core.IdenticalExpr (C, C++) 569
alpha.core.PointerArithm (C) 569
alpha.core.PointerSub (C) 570
alpha.core.SizeofPtr (C) 570
alpha.core.StackAddressAsyncEscape (C) 570
alpha.core.TestAfterDivZero (C) 570
alpha.cplusplus 570
alpha.cplusplus.DeleteWithNonVirtualDtor (C++) 570
alpha.cplusplus.EnumCastOutOfRange (C++) 571
alpha.cplusplus.InvalidatedIterator (C++) 571
alpha.cplusplus.IteratorRange (C++) 571
alpha.cplusplus.MismatchedIterator (C++) 571
alpha.cplusplus.MisusedMovedObject (C++) 572
alpha.cplusplus.SmartPtr (C++) 572
alpha.deadcode 572
alpha.deadcode.UnreachableCode (C, C++) 572
alpha.fuchsia 572
alpha.fuchsia.Lock 572
alpha.llvm 573
alpha.llvm.Conventions 573
alpha.osx 573
alpha.osx.cocoa.DirectIvarAssignment (ObjC) 573
alpha.osx.cocoa.DirectIvarAssignmentForAnnotatedFunctions (ObjC) 573
alpha.osx.cocoa.InstanceVariableInvalidation (ObjC) 573
alpha.osx.cocoa.MissingInvalidationMethod (ObjC) 574
alpha.osx.cocoa.localizability.PluralMisuseChecker (ObjC) 574

alpha.security 575
alpha.security.ArrayBound (C) 575
alpha.security.ArrayBoundV2 (C) 575
alpha.security.MallocOverflow (C) 576
alpha.security.MmapWriteExec (C) 576
alpha.security.ReturnPtrRange (C) 576
alpha.security.cert 577
alpha.security.cert.pos 577
alpha.security.cert.pos.34c 577
alpha.security.cert.env 577
alpha.security.cert.env.InvalidPtr 577
alpha.security.taint 578
alpha.security.taint.TaintPropagation (C, C++) 578
alpha.unix 579
alpha.unix.StdCLibraryFunctionArgs (C) 579
alpha.unix.BlockInCriticalSection (C) 580
alpha.unix.Chroot (C) 580
alpha.unix.PthreadLock (C) 580
alpha.unix.SimpleStream (C) 581
alpha.unix.Stream (C) 581
alpha.unix.cstring.BufferOverlap (C) 582
alpha.unix.cstring.NotNullTerminated (C) 582
alpha.unix.cstring.OutOfBounds (C) 582
alpha.unix.cstring.UninitializedRead (C) 582
alpha.nondeterminism.PointerIteration (C++) 583
alpha.nondeterminism.PointerSorting (C++) 583
alpha.WebKit 583
alpha.webkit.UncountedCallArgsChecker 583
alpha.webkit.UncountedLocalVarsChecker 584
Debug Checkers 585
debug 585
debug.AnalysisOrder 585
debug.ConfigDumper 585
debug.DumpCFG Display 585
debug.DumpCallGraph 585
debug.DumpCalls 586
debug.DumpDominators 586
debug.DumpLiveVars 586
debug.DumpTraversal 586
debug.ExprInspection 586
debug.Stats 586
debug.TaintTest 586
debug.ViewCFG 586
debug.ViewCallGraph 586
debug.ViewExplodedGraph 586

Security 691
Security **694**

Default Checkers

core

Models core language features and contains general-purpose checkers such as division by zero, null pointer dereference, usage of uninitialized values, etc. *These checkers must be always switched on as other checker rely on them.*

core.CallAndMessage (C, C++, ObjC)

Check for logical errors for function calls and Objective-C message expressions (e.g., uninitialized arguments, null function pointers).

```
//C
void test() {
   void (*foo)(void);
   foo = 0;
   foo(); // warn: function pointer is null
}

// C++
class C {
public:
  void f();
};

void test() {
  C *pc;
  pc->f(); // warn: object pointer is uninitialized
}

// C++
class C {
public:
  void f();
};

void test() {
  C *pc = 0;
  pc->f(); // warn: object pointer is null
}

// Objective-C
@interface MyClass : NSObject
@property (readwrite,assign) id x;
- (long double)longDoubleM;
@end

void test() {
  MyClass *obj1;
  long double ld1 = [obj1 longDoubleM];
    // warn: receiver is uninitialized
}

// Objective-C
@interface MyClass : NSObject
```

```
@property (readwrite,assign) id x;
- (long double)longDoubleM;
@end

void test() {
  MyClass *obj1;
  id i = obj1.x; // warn: uninitialized object pointer
}

// Objective-C
@interface Subscriptable : NSObject
- (id)objectAtIndexedSubscript:(unsigned int)index;
@end

@interface MyClass : Subscriptable
@property (readwrite,assign) id x;
- (long double)longDoubleM;
@end

void test() {
  MyClass *obj1;
  id i = obj1[0]; // warn: uninitialized object pointer
}
```

core.DivideZero (C, C++, ObjC)

Check for division by zero.

```
void test(int z) {
  if (z == 0)
    int x = 1 / z; // warn
}

void test() {
  int x = 1;
  int y = x % 0; // warn
}
```

core.NonNullParamChecker (C, C++, ObjC)

Check for null pointers passed as arguments to a function whose arguments are references or marked with the ‘nonnull’ attribute.

```
int f(int *p) __attribute__((nonnull));

void test(int *p) {
  if (!p)
    f(p); // warn
}
```

core.NullDereference (C, C++, ObjC)

Check for dereferences of null pointers.

This checker specifically does not report null pointer dereferences for x86 and x86-64 targets when the address space is 256 (x86 GS Segment), 257 (x86 FS Segment), or 258 (x86 SS segment). See X86/X86-64 Language Extensions for reference.

The `SuppressAddressSpaces` option suppresses warnings for null dereferences of all pointers with address spaces. You can disable this behavior with the option `-analyzer-config core.NullDereference:SuppressAddressSpaces=false`. *Defaults to true*.

```c
// C
void test(int *p) {
  if (p)
    return;

  int x = p[0]; // warn
}

// C
void test(int *p) {
  if (!p)
    *p = 0; // warn
}

// C++
class C {
public:
  int x;
};

void test() {
  C *pc = 0;
  int k = pc->x; // warn
}

// Objective-C
@interface MyClass {
@public
  int x;
}
@end

void test() {
  MyClass *obj = 0;
  obj->x = 1; // warn
}
```

core.StackAddressEscape (C)

Check that addresses to stack memory do not escape the function.

```c
char const *p;

void test() {
  char const str[] = "string";
  p = str; // warn
}

void* test() {
   return __builtin_alloca(12); // warn
}

void test() {
  static int *x;
  int y;
  x = &y; // warn
}
```

core.UndefinedBinaryOperatorResult (C)

Check for undefined results of binary operators.

```
void test() {
  int x;
  int y = x + 1; // warn: left operand is garbage
}
```

core.VLASize (C)

Check for declarations of Variable Length Arrays of undefined or zero size.

Check for declarations of VLA of undefined or zero size.

```
void test() {
  int x;
  int vla1[x]; // warn: garbage as size
}

void test() {
  int x = 0;
  int vla2[x]; // warn: zero size
}
```

core.uninitialized.ArraySubscript (C)

Check for uninitialized values used as array subscripts.

```
void test() {
  int i, a[10];
  int x = a[i]; // warn: array subscript is undefined
}
```

core.uninitialized.Assign (C)

Check for assigning uninitialized values.

```
void test() {
  int x;
  x |= 1; // warn: left expression is uninitialized
}
```

core.uninitialized.Branch (C)

Check for uninitialized values used as branch conditions.

```
void test() {
  int x;
  if (x) // warn
    return;
}
```

core.uninitialized.CapturedBlockVariable (C)

Check for blocks that capture uninitialized values.

```
void test() {
  int x;
  ^{ int y = x; }(); // warn
}
```

core.uninitialized.UndefReturn (C)

Check for uninitialized values being returned to the caller.

```
int test() {
  int x;
```

```
  return x; // warn
}
```

cplusplus

C++ Checkers.

cplusplus.InnerPointer (C++)

Check for inner pointers of C++ containers used after re/deallocation.

Many container methods in the C++ standard library are known to invalidate “references” (including actual references, iterators and raw pointers) to elements of the container. Using such references after they are invalidated causes undefined behavior, which is a common source of memory errors in C++ that this checker is capable of finding.

The checker is currently limited to `std::string` objects and doesn’t recognize some of the more sophisticated approaches to passing unowned pointers around, such as `std::string_view`.

```
void deref_after_assignment() {
  std::string s = "llvm";
  const char *c = s.data(); // note: pointer to inner buffer of 'std::string' obtained here
  s = "clang"; // note: inner buffer of 'std::string' reallocated by call to 'operator='
  consume(c); // warn: inner pointer of container used after re/deallocation
}

const char *return_temp(int x) {
  return std::to_string(x).c_str(); // warn: inner pointer of container used after re/deallocation
  // note: pointer to inner buffer of 'std::string' obtained here
  // note: inner buffer of 'std::string' deallocated by call to destructor
}
```

cplusplus.NewDelete (C++)

Check for double-free and use-after-free problems. Traces memory managed by new/delete.

```
void f(int *p);

void testUseMiddleArgAfterDelete(int *p) {
  delete p;
  f(p); // warn: use after free
}

class SomeClass {
public:
  void f();
};

void test() {
  SomeClass *c = new SomeClass;
  delete c;
  c->f(); // warn: use after free
}

void test() {
  int *p = (int *)__builtin_alloca(sizeof(int));
  delete p; // warn: deleting memory allocated by alloca
}

void test() {
  int *p = new int;
  delete p;
  delete p; // warn: attempt to free released
}
```

```
void test() {
  int i;
  delete &i; // warn: delete address of local
}

void test() {
  int *p = new int[1];
  delete[] (++p);
    // warn: argument to 'delete[]' is offset by 4 bytes
    // from the start of memory allocated by 'new[]'
}
```

cplusplus.NewDeleteLeaks (C++)

Check for memory leaks. Traces memory managed by new/delete.

```
void test() {
  int *p = new int;
} // warn
```

cplusplus.PlacementNewChecker (C++)

Check if default placement new is provided with pointers to sufficient storage capacity.

```
#include <new>

void f() {
  short s;
  long *lp = ::new (&s) long; // warn
}
```

cplusplus.SelfAssignment (C++)

Checks C++ copy and move assignment operators for self assignment.

cplusplus.StringChecker (C++)

Checks std::string operations.

Checks if the cstring pointer from which the `std::string` object is constructed is `NULL` or not. If the checker cannot reason about the nullness of the pointer it will assume that it was non-null to satisfy the precondition of the constructor.

This checker is capable of checking the SEI CERT C++ coding rule STR51-CPP. Do not attempt to create a std::string from a null pointer.

```
#include <string>

void f(const char *p) {
  if (!p) {
    std::string msg(p); // warn: The parameter must not be null
  }
}
```

deadcode

Dead Code Checkers.

deadcode.DeadStores (C)

Check for values stored to variables that are never read afterwards.

```
void test() {
  int x;
  x = 1; // warn
}
```

The `WarnForDeadNestedAssignments` option enables the checker to emit warnings for nested dead assignments. You can disable with the `-analyzer-config deadcode.DeadStores:WarnForDeadNestedAssignments=false`. *Defaults to true*.

Would warn for this e.g.: if ((y = make_int())) { }

nullability

Objective C checkers that warn for null pointer passing and dereferencing errors.

nullability.NullPassedToNonnull (ObjC)

Warns when a null pointer is passed to a pointer which has a _Nonnull type.

```
if (name != nil)
  return;
// Warning: nil passed to a callee that requires a non-null 1st parameter
NSString *greeting = [@"Hello " stringByAppendingString:name];
```

nullability.NullReturnedFromNonnull (ObjC)

Warns when a null pointer is returned from a function that has _Nonnull return type.

```
- (nonnull id)firstChild {
  id result = nil;
  if ([_children count] > 0)
    result = _children[0];

  // Warning: nil returned from a method that is expected
  // to return a non-null value
  return result;
}
```

nullability.NullableDereferenced (ObjC)

Warns when a nullable pointer is dereferenced.

```
struct LinkedList {
  int data;
  struct LinkedList *next;
};

struct LinkedList * _Nullable getNext(struct LinkedList *l);

void updateNextData(struct LinkedList *list, int newData) {
  struct LinkedList *next = getNext(list);
  // Warning: Nullable pointer is dereferenced
  next->data = 7;
}
```

nullability.NullablePassedToNonnull (ObjC)

Warns when a nullable pointer is passed to a pointer which has a _Nonnull type.

```
typedef struct Dummy { int val; } Dummy;
Dummy *_Nullable returnsNullable();
void takesNonnull(Dummy *_Nonnull);
```

```
void test() {
  Dummy *p = returnsNullable();
  takesNonnull(p); // warn
}
```

nullability.NullableReturnedFromNonnull (ObjC)

Warns when a nullable pointer is returned from a function that has _Nonnull return type.

optin

Checkers for portability, performance or coding style specific rules.

optin.cplusplus.UninitializedObject (C++)

This checker reports uninitialized fields in objects created after a constructor call. It doesn't only find direct uninitialized fields, but rather makes a deep inspection of the object, analyzing all of its fields' subfields. The checker regards inherited fields as direct fields, so one will receive warnings for uninitialized inherited data members as well.

```
// With Pedantic and CheckPointeeInitialization set to true

struct A {
  struct B {
    int x; // note: uninitialized field 'this->b.x'
    // note: uninitialized field 'this->bptr->x'
    int y; // note: uninitialized field 'this->b.y'
    // note: uninitialized field 'this->bptr->y'
  };
  int *iptr; // note: uninitialized pointer 'this->iptr'
  B b;
  B *bptr;
  char *cptr; // note: uninitialized pointee 'this->cptr'

  A (B *bptr, char *cptr) : bptr(bptr), cptr(cptr) {}
};

void f() {
  A::B b;
  char c;
  A a(&b, &c); // warning: 6 uninitialized fields
 //          after the constructor call
}

// With Pedantic set to false and
// CheckPointeeInitialization set to true
// (every field is uninitialized)

struct A {
  struct B {
    int x;
    int y;
  };
  int *iptr;
  B b;
  B *bptr;
  char *cptr;

  A (B *bptr, char *cptr) : bptr(bptr), cptr(cptr) {}
};

void f() {
```

```
  A::B b;
  char c;
  A a(&b, &c); // no warning
}

// With Pedantic set to true and
// CheckPointeeInitialization set to false
// (pointees are regarded as initialized)

struct A {
  struct B {
    int x; // note: uninitialized field 'this->b.x'
    int y; // note: uninitialized field 'this->b.y'
  };
  int *iptr; // note: uninitialized pointer 'this->iptr'
  B b;
  B *bptr;
  char *cptr;

  A (B *bptr, char *cptr) : bptr(bptr), cptr(cptr) {}
};

void f() {
  A::B b;
  char c;
  A a(&b, &c); // warning: 3 uninitialized fields
 //          after the constructor call
}
```

Options

This checker has several options which can be set from command line (e.g. `-analyzer-config optin.cplusplus.UninitializedObject:Pedantic=true`):

- `Pedantic` (boolean). If to false, the checker won't emit warnings for objects that don't have at least one initialized field. Defaults to false.
- `NotesAsWarnings` (boolean). If set to true, the checker will emit a warning for each uninitialized field, as opposed to emitting one warning per constructor call, and listing the uninitialized fields that belongs to it in notes. *Defaults to false*.
- `CheckPointeeInitialization` (boolean). If set to false, the checker will not analyze the pointee of pointer/reference fields, and will only check whether the object itself is initialized. *Defaults to false*.
- `IgnoreRecordsWithField` (string). If supplied, the checker will not analyze structures that have a field with a name or type name that matches the given pattern. *Defaults to ""*.

optin.cplusplus.VirtualCall (C++)

Check virtual function calls during construction or destruction.

```
class A {
public:
  A() {
    f(); // warn
  }
  virtual void f();
};

class A {
public:
  ~A() {
    this->f(); // warn
```

```
  }
  virtual void f();
};
```

optin.mpi.MPI-Checker (C)

Checks MPI code.

```
void test() {
  double buf = 0;
  MPI_Request sendReq1;
  MPI_Ireduce(MPI_IN_PLACE, &buf, 1, MPI_DOUBLE, MPI_SUM,
      0, MPI_COMM_WORLD, &sendReq1);
} // warn: request 'sendReq1' has no matching wait.

void test() {
  double buf = 0;
  MPI_Request sendReq;
  MPI_Isend(&buf, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &sendReq);
  MPI_Irecv(&buf, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &sendReq); // warn
  MPI_Isend(&buf, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &sendReq); // warn
  MPI_Wait(&sendReq, MPI_STATUS_IGNORE);
}

void missingNonBlocking() {
  int rank = 0;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Request sendReq1[10][10][10];
  MPI_Wait(&sendReq1[1][7][9], MPI_STATUS_IGNORE); // warn
}
```

optin.osx.cocoa.localizability.EmptyLocalizationContextChecker (ObjC)

Check that NSLocalizedString macros include a comment for context.

```
- (void)test {
  NSString *string = NSLocalizedString(@"LocalizedString", nil); // warn
  NSString *string2 = NSLocalizedString(@"LocalizedString", @" "); // warn
  NSString *string3 = NSLocalizedStringWithDefaultValue(
    @"LocalizedString", nil, [[NSBundle alloc] init], nil,@""); // warn
}
```

optin.osx.cocoa.localizability.NonLocalizedStringChecker (ObjC)

Warns about uses of non-localized NSStrings passed to UI methods expecting localized NSStrings.

```
NSString *alarmText =
  NSLocalizedString(@"Enabled", @"Indicates alarm is turned on");
if (!isEnabled) {
  alarmText = @"Disabled";
}
UILabel *alarmStateLabel = [[UILabel alloc] init];

// Warning: User-facing text should use localized string macro
[alarmStateLabel setText:alarmText];
```

optin.performance.GCDAntipattern

Check for performance anti-patterns when using Grand Central Dispatch.

optin.performance.Padding

Check for excessively padded structs.

optin.portability.UnixAPI

Finds implementation-defined behavior in UNIX/Posix functions.

security

Security related checkers.

security.FloatLoopCounter (C)

Warn on using a floating point value as a loop counter (CERT: FLP30-C, FLP30-CPP).

```
void test() {
  for (float x = 0.1f; x <= 1.0f; x += 0.1f) {} // warn
}
```

security.insecureAPI.UncheckedReturn (C)

Warn on uses of functions whose return values must be always checked.

```
void test() {
  setuid(1); // warn
}
```

security.insecureAPI.bcmp (C)

Warn on uses of the ‘bcmp’ function.

```
void test() {
  bcmp(ptr0, ptr1, n); // warn
}
```

security.insecureAPI.bcopy (C)

Warn on uses of the ‘bcopy’ function.

```
void test() {
  bcopy(src, dst, n); // warn
}
```

security.insecureAPI.bzero (C)

Warn on uses of the ‘bzero’ function.

```
void test() {
  bzero(ptr, n); // warn
}
```

security.insecureAPI.getpw (C)

Warn on uses of the ‘getpw’ function.

```
void test() {
  char buff[1024];
  getpw(2, buff); // warn
}
```

security.insecureAPI.gets (C)

Warn on uses of the ‘gets’ function.

```
void test() {
  char buff[1024];
  gets(buff); // warn
}
```

security.insecureAPI.mkstemp (C)

Warn when ‘mkstemp’ is passed fewer than 6 X’s in the format string.

```
void test() {
  mkstemp("XX"); // warn
}
```

security.insecureAPI.mktemp (C)

Warn on uses of the `mktemp` function.

```
void test() {
  char *x = mktemp("/tmp/zxcv"); // warn: insecure, use mkstemp
}
```

security.insecureAPI.rand (C)

Warn on uses of inferior random number generating functions (only if arc4random function is available): `drand48, erand48, jrand48, lcong48, lrand48, mrand48, nrand48, random, rand_r`.

```
void test() {
  random(); // warn
}
```

security.insecureAPI.strcpy (C)

Warn on uses of the `strcpy` and `strcat` functions.

```
void test() {
  char x[4];
  char *y = "abcd";

  strcpy(x, y); // warn
}
```

security.insecureAPI.vfork (C)

Warn on uses of the ‘vfork’ function.

```
void test() {
  vfork(); // warn
}
```

security.insecureAPI.DeprecatedOrUnsafeBufferHandling (C)

unsafe or deprecated buffer handling functions, which `vscanf, vwscanf, vfscanf, vfwscanf, sscanf, swscanf, vsscanf, vswscanf, swprintf, snprintf, vswprint`

```
void test() {
  char buf [5];
  strncpy(buf, "a", 1); // warn
}
```

unix

POSIX/Unix checkers.

unix.API (C)

Check calls to various UNIX/Posix functions: `open, pthread_once, calloc, malloc, realloc, alloca`.

```
// Currently the check is performed for apple targets only.
void test(const char *path) {
  int fd = open(path, O_CREAT);
    // warn: call to 'open' requires a third argument when the
    // 'O_CREAT' flag is set
}

void f();

void test() {
  pthread_once_t pred = {0x30B1BCBA, {0}};
  pthread_once(&pred, f);
    // warn: call to 'pthread_once' uses the local variable
}

void test() {
  void *p = malloc(0); // warn: allocation size of 0 bytes
}

void test() {
  void *p = calloc(0, 42); // warn: allocation size of 0 bytes
}

void test() {
  void *p = malloc(1);
  p = realloc(p, 0); // warn: allocation size of 0 bytes
}

void test() {
  void *p = alloca(0); // warn: allocation size of 0 bytes
}

void test() {
  void *p = valloc(0); // warn: allocation size of 0 bytes
}
```

unix.Malloc (C)

Check for memory leaks, double free, and use-after-free problems. Traces memory managed by malloc()/free().

```
void test() {
  int *p = malloc(1);
  free(p);
  free(p); // warn: attempt to free released memory
}

void test() {
  int *p = malloc(sizeof(int));
  free(p);
  *p = 1; // warn: use after free
}

void test() {
  int *p = malloc(1);
  if (p)
    return; // warn: memory is never released
}
```

```
void test() {
  int a[] = { 1 };
  free(a); // warn: argument is not allocated by malloc
}

void test() {
  int *p = malloc(sizeof(char));
  p = p - 1;
  free(p); // warn: argument to free() is offset by -4 bytes
}
```

unix.MallocSizeof (C)

Check for dubious `malloc` arguments involving `sizeof`.

```
void test() {
  long *p = malloc(sizeof(short));
    // warn: result is converted to 'long *', which is
    // incompatible with operand type 'short'
  free(p);
}
```

unix.MismatchedDeallocator (C, C++)

Check for mismatched deallocators.

```
// C, C++
void test() {
  int *p = (int *)malloc(sizeof(int));
  delete p; // warn
}

// C, C++
void __attribute((ownership_returns(malloc))) *user_malloc(size_t);

void test() {
  int *p = (int *)user_malloc(sizeof(int));
  delete p; // warn
}

// C, C++
void test() {
  int *p = new int;
  free(p); // warn
}

// C, C++
void test() {
  int *p = new int[1];
  realloc(p, sizeof(long)); // warn
}

// C, C++
template <typename T>
struct SimpleSmartPointer {
  T *ptr;

  explicit SimpleSmartPointer(T *p = 0) : ptr(p) {}
  ~SimpleSmartPointer() {
    delete ptr; // warn
  }
```

```
};

void test() {
  SimpleSmartPointer<int> a((int *)malloc(4));
}

// C++
void test() {
  int *p = (int *)operator new(0);
  delete[] p; // warn
}

// Objective-C, C++
void test(NSUInteger dataLength) {
  int *p = new int;
  NSData *d = [NSData dataWithBytesNoCopy:p
                length:sizeof(int) freeWhenDone:1];
    // warn +dataWithBytesNoCopy:length:freeWhenDone: cannot take
    // ownership of memory allocated by 'new'
}
```

unix.Vfork (C)

Check for proper usage of `vfork`.

```
int test(int x) {
  pid_t pid = vfork(); // warn
  if (pid != 0)
    return 0;

  switch (x) {
  case 0:
    pid = 1;
    execl("", "", 0);
    _exit(1);
    break;
  case 1:
    x = 0; // warn: this assignment is prohibited
    break;
  case 2:
    foo(); // warn: this function call is prohibited
    break;
  default:
    return 0; // warn: return is prohibited
  }

  while(1);
}
```

unix.cstring.BadSizeArg (C)

Check the size argument passed into C string functions for common erroneous patterns. Use `-Wno-strncat-size` compiler option to mute other `strncat`-related compiler warnings.

```
void test() {
  char dest[3];
  strncat(dest, """""""""""""""""""""""""""""*", sizeof(dest));
    // warn: potential buffer overflow
}
```

unix.cstring.NullArg (C)

Check for null pointers being passed as arguments to C string functions: `strlen, strnlen, strcpy, strncpy, strcat, strncat, strcmp, strncmp, strcasecmp, strncasecmp`.

```
int test() {
  return strlen(0); // warn
}
```

osx

macOS checkers.

osx.API (C)

Check for proper uses of various Apple APIs.

```
void test() {
  dispatch_once_t pred = 0;
  dispatch_once(&pred, ^(){}); // warn: dispatch_once uses local
}
```

osx.NumberObjectConversion (C, C++, ObjC)

Check for erroneous conversions of objects representing numbers into numbers.

```
NSNumber *photoCount = [albumDescriptor objectForKey:@"PhotoCount"];
// Warning: Comparing a pointer value of type 'NSNumber *'
// to a scalar integer value
if (photoCount > 0) {
  [self displayPhotos];
}
```

osx.ObjCProperty (ObjC)

Check for proper uses of Objective-C properties.

```
NSNumber *photoCount = [albumDescriptor objectForKey:@"PhotoCount"];
// Warning: Comparing a pointer value of type 'NSNumber *'
// to a scalar integer value
if (photoCount > 0) {
  [self displayPhotos];
}
```

osx.SecKeychainAPI (C)

Check for proper uses of Secure Keychain APIs.

```
void test() {
  unsigned int *ptr = 0;
  UInt32 length;

  SecKeychainItemFreeContent(ptr, &length);
    // warn: trying to free data which has not been allocated
}

void test() {
  unsigned int *ptr = 0;
  UInt32 *length = 0;
  void *outData;

  OSStatus st =
    SecKeychainItemCopyContent(2, ptr, ptr, length, outData);
    // warn: data is not released
```

```
}

void test() {
  unsigned int *ptr = 0;
  UInt32 *length = 0;
  void *outData;

  OSStatus st =
    SecKeychainItemCopyContent(2, ptr, ptr, length, &outData);

  SecKeychainItemFreeContent(ptr, outData);
    // warn: only call free if a non-NULL buffer was returned
}

void test() {
  unsigned int *ptr = 0;
  UInt32 *length = 0;
  void *outData;

  OSStatus st =
    SecKeychainItemCopyContent(2, ptr, ptr, length, &outData);

  st = SecKeychainItemCopyContent(2, ptr, ptr, length, &outData);
    // warn: release data before another call to the allocator

  if (st == noErr)
    SecKeychainItemFreeContent(ptr, outData);
}

void test() {
  SecKeychainItemRef itemRef = 0;
  SecKeychainAttributeInfo *info = 0;
  SecItemClass *itemClass = 0;
  SecKeychainAttributeList *attrList = 0;
  UInt32 *length = 0;
  void *outData = 0;

  OSStatus st =
    SecKeychainItemCopyAttributesAndData(itemRef, info,
                                         itemClass, &attrList,
                                         length, &outData);

  SecKeychainItemFreeContent(attrList, outData);
    // warn: deallocator doesn't match the allocator
}
```

osx.cocoa.AtSync (ObjC)

Check for nil pointers used as mutexes for @synchronized.

```
void test(id x) {
  if (!x)
    @synchronized(x) {} // warn: nil value used as mutex
}

void test() {
  id y;
  @synchronized(y) {} // warn: uninitialized value used as mutex
}
```

osx.cocoa.AutoreleaseWrite

Warn about potentially crashing writes to autoreleasing objects from different autoreleasing pools in Objective-C.

osx.cocoa.ClassRelease (ObjC)

Check for sending ‘retain’, ‘release’, or ‘autorelease’ directly to a Class.

```
@interface MyClass : NSObject
@end

void test(void) {
  [MyClass release]; // warn
}
```

osx.cocoa.Dealloc (ObjC)

Warn about Objective-C classes that lack a correct implementation of -dealloc

```
@interface MyObject : NSObject {
  id _myproperty;
}
@end

@implementation MyObject // warn: lacks 'dealloc'
@end

@interface MyObject : NSObject {}
@property(assign) id myproperty;
@end

@implementation MyObject // warn: does not send 'dealloc' to super
- (void)dealloc {
  self.myproperty = 0;
}
@end

@interface MyObject : NSObject {
  id _myproperty;
}
@property(retain) id myproperty;
@end

@implementation MyObject
@synthesize myproperty = _myproperty;
  // warn: var was retained but wasn't released
- (void)dealloc {
  [super dealloc];
}
@end

@interface MyObject : NSObject {
  id _myproperty;
}
@property(assign) id myproperty;
@end

@implementation MyObject
@synthesize myproperty = _myproperty;
  // warn: var wasn't retained but was released
- (void)dealloc {
  [_myproperty release];
```

```
  [super dealloc];
}
@end
```

osx.cocoa.IncompatibleMethodTypes (ObjC)

Warn about Objective-C method signatures with type incompatibilities.

```
@interface MyClass1 : NSObject
- (int)foo;
@end

@implementation MyClass1
- (int)foo { return 1; }
@end

@interface MyClass2 : MyClass1
- (float)foo;
@end

@implementation MyClass2
- (float)foo { return 1.0; } // warn
@end
```

osx.cocoa.Loops

Improved modeling of loops using Cocoa collection types.

osx.cocoa.MissingSuperCall (ObjC)

Warn about Objective-C methods that lack a necessary call to super.

```
@interface Test : UIViewController
@end
@implementation test
- (void)viewDidLoad {} // warn
@end
```

osx.cocoa.NSAutoreleasePool (ObjC)

Warn for suboptimal uses of NSAutoreleasePool in Objective-C GC mode.

```
void test() {
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  [pool release]; // warn
}
```

osx.cocoa.NSError (ObjC)

Check usage of NSError parameters.

```
@interface A : NSObject
- (void)foo:(NSError """""""""""""""""""""""""")error;
@end

@implementation A
- (void)foo:(NSError """""""""""""""""""""""""")error {
  // warn: method accepting NSError""""""""""""""""""""""""" should have a non-void
  // return value
}
@end

@interface A : NSObject
```

```
- (BOOL)foo:(NSError """""""""""""""""""""""""")error;
@end

@implementation A
- (BOOL)foo:(NSError """""""""""""""""""""""""")error {
  *error = 0; // warn: potential null dereference
  return 0;
}
@end
```

osx.cocoa.NilArg (ObjC)

Check for prohibited nil arguments to ObjC method calls.

- caseInsensitiveCompare:
- compare:
- compare:options:
- compare:options:range:
- compare:options:range:locale:
- componentsSeparatedByCharactersInSet:
- initWithFormat:

```
NSComparisonResult test(NSString *s) {
  NSString *aString = nil;
  return [s caseInsensitiveCompare:aString];
    // warn: argument to 'NSString' method
    // 'caseInsensitiveCompare:' cannot be nil
}
```

osx.cocoa.NonNilReturnValue

Models the APIs that are guaranteed to return a non-nil value.

osx.cocoa.ObjCGenerics (ObjC)

Check for type errors when using Objective-C generics.

```
NSMutableArray *names = [NSMutableArray array];
NSMutableArray *birthDates = names;

// Warning: Conversion from value of type 'NSDate *'
// to incompatible type 'NSString *'
[birthDates addObject: [NSDate date]];
```

osx.cocoa.RetainCount (ObjC)

Check for leaks and improper reference count management

```
void test() {
  NSString *s = [[NSString alloc] init]; // warn
}

CFStringRef test(char *bytes) {
  return CFStringCreateWithCStringNoCopy(
            0, bytes, NSNEXTSTEPStringEncoding, 0); // warn
}
```

osx.cocoa.RunLoopAutoreleaseLeak

Check for leaked memory in autorelease pools that will never be drained.

osx.cocoa.SelfInit (ObjC)

Check that ‘self’ is properly initialized inside an initializer method.

```
@interface MyObj : NSObject {
  id x;
}
- (id)init;
@end

@implementation MyObj
- (id)init {
  [super init];
  x = 0; // warn: instance variable used while 'self' is not
         // initialized
  return 0;
}
@end

@interface MyObj : NSObject
- (id)init;
@end

@implementation MyObj
- (id)init {
  [super init];
  return self; // warn: returning uninitialized 'self'
}
@end
```

osx.cocoa.SuperDealloc (ObjC)

Warn about improper use of ‘[super dealloc]’ in Objective-C.

```
@interface SuperDeallocThenReleaseIvarClass : NSObject {
  NSObject *_ivar;
}
@end

@implementation SuperDeallocThenReleaseIvarClass
- (void)dealloc {
  [super dealloc];
  [_ivar release]; // warn
}
@end
```

osx.cocoa.UnusedIvars (ObjC)

Warn about private ivars that are never used.

```
@interface MyObj : NSObject {
@private
  id x; // warn
}
@end

@implementation MyObj
@end
```

osx.cocoa.VariadicMethodTypes (ObjC)

Check for passing non-Objective-C types to variadic collection initialization methods that expect only Objective-C types.

```
void test() {
  [NSSet setWithObjects:@"Foo", "Bar", nil];
    // warn: argument should be an ObjC pointer type, not 'char *'
}
```

osx.coreFoundation.CFError (C)

Check usage of CFErrorRef* parameters

```
void test(CFErrorRef *error) {
  // warn: function accepting CFErrorRef* should have a
  // non-void return
}

int foo(CFErrorRef *error) {
  *error = 0; // warn: potential null dereference
  return 0;
}
```

osx.coreFoundation.CFNumber (C)

Check for proper uses of CFNumber APIs.

```
CFNumberRef test(unsigned char x) {
  return CFNumberCreate(0, kCFNumberSInt16Type, &x);
   // warn: 8 bit integer is used to initialize a 16 bit integer
}
```

osx.coreFoundation.CFRetainRelease (C)

Check for null arguments to CFRetain/CFRelease/CFMakeCollectable.

```
void test(CFTypeRef p) {
  if (!p)
    CFRetain(p); // warn
}

void test(int x, CFTypeRef p) {
  if (p)
    return;

  CFRelease(p); // warn
}
```

osx.coreFoundation.containers.OutOfBounds (C)

Checks for index out-of-bounds when using ‘CFArray’ API.

```
void test() {
  CFArrayRef A = CFArrayCreate(0, 0, 0, &kCFTypeArrayCallBacks);
  CFArrayGetValueAtIndex(A, 0); // warn
}
```

osx.coreFoundation.containers.PointerSizedValues (C)

Warns if ‘CFArray’, ‘CFDictionary’, ‘CFSet’ are created with non-pointer-size values.

```
void test() {
  int x[] = { 1 };
```

```
  CFArrayRef A = CFArrayCreate(0, (const void """""""""""""""""""""""""")x, 1,
                               &kCFTypeArrayCallBacks); // warn
}
```

Fuchsia

Fuchsia is an open source capability-based operating system currently being developed by Google. This section describes checkers that can find various misuses of Fuchsia APIs.

fuchsia.HandleChecker

Handles identify resources. Similar to pointers they can be leaked, double freed, or use after freed. This check attempts to find such problems.

```
void checkLeak08(int tag) {
  zx_handle_t sa, sb;
  zx_channel_create(0, &sa, &sb);
  if (tag)
    zx_handle_close(sa);
  use(sb); // Warn: Potential leak of handle
  zx_handle_close(sb);
}
```

WebKit

WebKit is an open-source web browser engine available for macOS, iOS and Linux. This section describes checkers that can find issues in WebKit codebase.

Most of the checkers focus on memory management for which WebKit uses custom implementation of reference counted smartpointers.

Checkers are formulated in terms related to ref-counting:

- *Ref-counted type* is either `Ref<T>` or `RefPtr<T>`.
- *Ref-countable type* is any type that implements `ref()` and `deref()` methods as `RefPtr<>` is a template (i. e. relies on duck typing).
- *Uncounted type* is ref-countable but not ref-counted type.

webkit.RefCntblBaseVirtualDtor

All uncounted types used as base classes must have a virtual destructor.

Ref-counted types hold their ref-countable data by a raw pointer and allow implicit upcasting from ref-counted pointer to derived type to ref-counted pointer to base type. This might lead to an object of (dynamic) derived type being deleted via pointer to the base class type which C++ standard defines as UB in case the base class doesn't have virtual destructor `[expr.delete]`.

```
struct RefCntblBase {
  void ref() {}
  void deref() {}
};

struct Derived : RefCntblBase { }; // warn
```

webkit.NoUncountedMemberChecker

Raw pointers and references to uncounted types can't be used as class members. Only ref-counted types are allowed.

```
struct RefCntbl {
  void ref() {}
  void deref() {}
};
```

```
struct Foo {
  RefCntbl * ptr; // warn
  RefCntbl & ptr; // warn
  // ...
};
```

webkit.UncountedLambdaCapturesChecker

Raw pointers and references to uncounted types can't be captured in lambdas. Only ref-counted types are allowed.

```
struct RefCntbl {
  void ref() {}
  void deref() {}
};

void foo(RefCntbl* a, RefCntbl& b) {
  [&, a](){ // warn about 'a'
    do_something(b); // warn about 'b'
  };
};
```

Experimental Checkers

These are checkers with known issues or limitations that keep them from being on by default. They are likely to have false positives. Bug reports and especially patches are welcome.

alpha.clone

alpha.clone.CloneChecker (C, C++, ObjC)

Reports similar pieces of code.

```
void log();

int max(int a, int b) { // warn
  log();
  if (a > b)
    return a;
  return b;
}

int maxClone(int x, int y) { // similar code here
  log();
  if (x > y)
    return x;
  return y;
}
```

alpha.core

alpha.core.BoolAssignment (ObjC)

Warn about assigning non-{0,1} values to boolean variables.

```
void test() {
  BOOL b = -1; // warn
}
```

alpha.core.C11Lock

Similarly to alpha.unix.PthreadLock, checks for the locking/unlocking of `mtx_t` mutexes.

```
mtx_t mtx1;

void bad1(void)
{
  mtx_lock(&mtx1);
  mtx_lock(&mtx1); // warn: This lock has already been acquired
}
```

alpha.core.CallAndMessageUnInitRefArg (C,C++, ObjC)

Check for logical errors for function calls and Objective-C message expressions (e.g., uninitialized arguments, null function pointers, and pointer to undefined variables).

```
void test(void) {
  int t;
  int &p = t;
  int &s = p;
  int &q = s;
  foo(q); // warn
}

void test(void) {
  int x;
  foo(&x); // warn
}
```

alpha.core.CastSize (C)

Check when casting a malloc'ed type T, whether the size is a multiple of the size of T.

```
void test() {
  int *x = (int *) malloc(11); // warn
}
```

alpha.core.CastToStruct (C, C++)

Check for cast from non-struct pointer to struct pointer.

```
// C
struct s {};

void test(int *p) {
  struct s *ps = (struct s *) p; // warn
}

// C++
class c {};

void test(int *p) {
  c *pc = (c *) p; // warn
}
```

alpha.core.Conversion (C, C++, ObjC)

Loss of sign/precision in implicit conversions.

```
void test(unsigned U, signed S) {
  if (S > 10) {
    if (U < S) {
    }
```

```
  }
  if (S < -10) {
    if (U < S) { // warn (loss of sign)
    }
  }
}

void test() {
  long long A = 1LL << 60;
  short X = A; // warn (loss of precision)
}
```

alpha.core.DynamicTypeChecker (ObjC)

Check for cases where the dynamic and the static type of an object are unrelated.

```
id date = [NSDate date];

// Warning: Object has a dynamic type 'NSDate *' which is
// incompatible with static type 'NSNumber *'"
NSNumber *number = date;
[number doubleValue];
```

alpha.core.FixedAddr (C)

Check for assignment of a fixed address to a pointer.

```
void test() {
  int *p;
  p = (int *) 0x10000; // warn
}
```

alpha.core.IdenticalExpr (C, C++)

Warn about unintended use of identical expressions in operators.

```
// C
void test() {
  int a = 5;
  int b = a | 4 | a; // warn: identical expr on both sides
}

// C++
bool f(void);

void test(bool b) {
  int i = 10;
  if (f()) { // warn: true and false branches are identical
    do {
      i--;
    } while (f());
  } else {
    do {
      i--;
    } while (f());
  }
}
```

alpha.core.PointerArithm (C)

Check for pointer arithmetic on locations other than array elements.

```
void test() {
  int x;
  int *p;
  p = &x + 1; // warn
}
```

alpha.core.PointerSub (C)

Check for pointer subtractions on two pointers pointing to different memory chunks.

```
void test() {
  int x, y;
  int d = &y - &x; // warn
}
```

alpha.core.SizeofPtr (C)

Warn about unintended use of `sizeof()` on pointer expressions.

```
struct s {};

int test(struct s *p) {
  return sizeof(p);
    // warn: sizeof(ptr) can produce an unexpected result
}
```

alpha.core.StackAddressAsyncEscape (C)

Check that addresses to stack memory do not escape the function that involves dispatch_after or dispatch_async. This checker is a part of `core.StackAddressEscape`, but is temporarily disabled until some false positives are fixed.

```
dispatch_block_t test_block_inside_block_async_leak() {
  int x = 123;
  void (^inner)(void) = ^void(void) {
    int y = x;
    ++y;
  };
  void (^outer)(void) = ^void(void) {
    int z = x;
    ++z;
    inner();
  };
  return outer; // warn: address of stack-allocated block is captured by a
                //       returned block
}
```

alpha.core.TestAfterDivZero (C)

Check for division by variable that is later compared against 0. Either the comparison is useless or there is division by zero.

```
void test(int x) {
  var = 77 / x;
  if (x == 0) { } // warn
}
```

alpha.cplusplus

alpha.cplusplus.DeleteWithNonVirtualDtor (C++)

Reports destructions of polymorphic objects with a non-virtual destructor in their base class.

```
NonVirtual *create() {
  NonVirtual *x = new NVDerived(); // note: conversion from derived to base
                                   //       happened here
  return x;
}

void sink(NonVirtual *x) {
  delete x; // warn: destruction of a polymorphic object with no virtual
            //       destructor
}
```

alpha.cplusplus.EnumCastOutOfRange (C++)

Check for integer to enumeration casts that could result in undefined values.

```
enum TestEnum {
  A = 0
};

void foo() {
  TestEnum t = static_cast(-1);
      // warn: the value provided to the cast expression is not in
      //       the valid range of values for the enum
```

alpha.cplusplus.InvalidatedIterator (C++)

Check for use of invalidated iterators.

```
void bad_copy_assign_operator_list1(std::list &L1,
                                    const std::list &L2) {
  auto i0 = L1.cbegin();
  L1 = L2;
  *i0; // warn: invalidated iterator accessed
}
```

alpha.cplusplus.IteratorRange (C++)

Check for iterators used outside their valid ranges.

```
void simple_bad_end(const std::vector &v) {
  auto i = v.end();
  *i; // warn: iterator accessed outside of its range
}
```

alpha.cplusplus.MismatchedIterator (C++)

Check for use of iterators of different containers where iterators of the same container are expected.

```
void bad_insert3(std::vector &v1, std::vector &v2) {
  v2.insert(v1.cbegin(), v2.cbegin(), v2.cend()); // warn: container accessed
                                                  //       using foreign
                                                  //       iterator argument
  v1.insert(v1.cbegin(), v1.cbegin(), v2.cend()); // warn: iterators of
                                                  //       different containers
                                                  //       used where the same
                                                  //       container is
                                                  //       expected
  v1.insert(v1.cbegin(), v2.cbegin(), v1.cend()); // warn: iterators of
                                                  //       different containers
                                                  //       used where the same
                                                  //       container is
```

```
                                                    //       expected
}
```

alpha.cplusplus.MisusedMovedObject (C++)

Method calls on a moved-from object and copying a moved-from object will be reported.

```
 struct A {
  void foo() {}
};

void f() {
  A a;
  A b = std::move(a); // note: 'a' became 'moved-from' here
  a.foo();            // warn: method call on a 'moved-from' object 'a'
}
```

alpha.cplusplus.SmartPtr (C++)

Check for dereference of null smart pointers.

```
void deref_smart_ptr() {
  std::unique_ptr<int> P;
  *P; // warn: dereference of a default constructed smart unique_ptr
}
```

alpha.deadcode

alpha.deadcode.UnreachableCode (C, C++)

Check unreachable code.

```
// C
int test() {
  int x = 1;
  while(x);
  return x; // warn
}

// C++
void test() {
  int a = 2;

  while (a > 1)
    a--;

  if (a > 1)
    a++; // warn
}

// Objective-C
void test(id x) {
  return;
  [x retain]; // warn
}
```

alpha.fuchsia

alpha.fuchsia.Lock

Similarly to alpha.unix.PthreadLock, checks for the locking/unlocking of fuchsia mutexes.

```
spin_lock_t mtx1;

void bad1(void)
{
  spin_lock(&mtx1);
  spin_lock(&mtx1);    // warn: This lock has already been acquired
}
```

alpha.llvm

alpha.llvm.Conventions

Check code for LLVM codebase conventions:

- A StringRef should not be bound to a temporary std::string whose lifetime is shorter than the StringRef's.
- Clang AST nodes should not have fields that can allocate memory.

alpha.osx

alpha.osx.cocoa.DirectIvarAssignment (ObjC)

Check for direct assignments to instance variables.

```
@interface MyClass : NSObject {}
@property (readonly) id A;
- (void) foo;
@end

@implementation MyClass
- (void) foo {
  _A = 0; // warn
}
@end
```

alpha.osx.cocoa.DirectIvarAssignmentForAnnotatedFunctions (ObjC)

Check for direct assignments to instance variables in the methods annotated with `objc_no_direct_instance_variable_assignment`.

```
@interface MyClass : NSObject {}
@property (readonly) id A;
- (void) fAnnotated __attribute__((
    annotate("objc_no_direct_instance_variable_assignment")));
- (void) fNotAnnotated;
@end

@implementation MyClass
- (void) fAnnotated {
  _A = 0; // warn
}
- (void) fNotAnnotated {
  _A = 0; // no warn
}
@end
```

alpha.osx.cocoa.InstanceVariableInvalidation (ObjC)

Check that the invalidatable instance variables are invalidated in the methods annotated with objc_instance_variable_invalidator.

```
@protocol Invalidation <NSObject>
- (void) invalidate
  __attribute__((annotate("objc_instance_variable_invalidator")));
@end

@interface InvalidationImpObj : NSObject <Invalidation>
@end

@interface SubclassInvalidationImpObj : InvalidationImpObj {
  InvalidationImpObj *var;
}
- (void)invalidate;
@end

@implementation SubclassInvalidationImpObj
- (void) invalidate {}
@end
// warn: var needs to be invalidated or set to nil
```

alpha.osx.cocoa.MissingInvalidationMethod (ObjC)

Check that the invalidation methods are present in classes that contain invalidatable instance variables.

```
@protocol Invalidation <NSObject>
- (void)invalidate
  __attribute__((annotate("objc_instance_variable_invalidator")));
@end

@interface NeedInvalidation : NSObject <Invalidation>
@end

@interface MissingInvalidationMethodDecl : NSObject {
  NeedInvalidation *Var; // warn
}
@end

@implementation MissingInvalidationMethodDecl
@end
```

alpha.osx.cocoa.localizability.PluralMisuseChecker (ObjC)

Warns against using one vs. many plural pattern in code when generating localized strings.

```
NSString *reminderText =
  NSLocalizedString(@"None", @"Indicates no reminders");
if (reminderCount == 1) {
  // Warning: Plural cases are not supported across all languages.
  // Use a .stringsdict file instead
  reminderText =
    NSLocalizedString(@"1 Reminder", @"Indicates single reminder");
} else if (reminderCount >= 2) {
  // Warning: Plural cases are not supported across all languages.
  // Use a .stringsdict file instead
  reminderText =
    [NSString stringWithFormat:
      NSLocalizedString(@"%@ Reminders", @"Indicates multiple reminders"),
        reminderCount];
}
```

alpha.security

alpha.security.ArrayBound (C)

Warn about buffer overflows (older checker).

```
void test() {
  char *s = "";
  char c = s[1]; // warn
}

struct seven_words {
  int c[7];
};

void test() {
  struct seven_words a, *p;
  p = &a;
  p[0] = a;
  p[1] = a;
  p[2] = a; // warn
}

// note: requires unix.Malloc or
// alpha.unix.MallocWithAnnotations checks enabled.
void test() {
  int *p = malloc(12);
  p[3] = 4; // warn
}

void test() {
  char a[2];
  int *b = (int*)a;
  b[1] = 3; // warn
}
```

alpha.security.ArrayBoundV2 (C)

Warn about buffer overflows (newer checker).

```
void test() {
  char *s = "";
  char c = s[1]; // warn
}

void test() {
  int buf[100];
  int *p = buf;
  p = p + 99;
  p[1] = 1; // warn
}

// note: compiler has internal check for this.
// Use -Wno-array-bounds to suppress compiler warning.
void test() {
  int buf[100][100];
  buf[0][-1] = 1; // warn
}

// note: requires alpha.security.taint check turned on.
void test() {
  char s[] = "abc";
```

```
  int x = getchar();
  char c = s[x]; // warn: index is tainted
}
```

alpha.security.MallocOverflow (C)

Check for overflows in the arguments to `malloc()`. It tries to catch `malloc(n * c)` patterns, where:

- n: a variable or member access of an object
- c: a constant foldable integral

This checker was designed for code audits, so expect false-positive reports. One is supposed to silence this checker by ensuring proper bounds checking on the variable in question using e.g. an `assert()` or a branch.

```
void test(int n) {
  void *p = malloc(n * sizeof(int)); // warn
}

void test2(int n) {
  if (n > 100) // gives an upper-bound
    return;
  void *p = malloc(n * sizeof(int)); // no warning
}

void test3(int n) {
  assert(n <= 100 && "Contract violated.");
  void *p = malloc(n * sizeof(int)); // no warning
}
```

Limitations:

- The checker won't warn for variables involved in explicit casts, since that might limit the variable's domain. E.g.: `(unsigned char)int x` would limit the domain to `[0,255]`. The checker will miss the true-positive cases when the explicit cast would not tighten the domain to prevent the overflow in the subsequent multiplication operation.
- It is an AST-based checker, thus it does not make use of the path-sensitive taint-analysis.

alpha.security.MmapWriteExec (C)

Warn on mmap() calls that are both writable and executable.

```
void test(int n) {
  void *c = mmap(NULL, 32, PROT_READ | PROT_WRITE | PROT_EXEC,
                 MAP_PRIVATE | MAP_ANON, -1, 0);
  // warn: Both PROT_WRITE and PROT_EXEC flags are set. This can lead to
  //       exploitable memory regions, which could be overwritten with malicious
  //       code
}
```

alpha.security.ReturnPtrRange (C)

Check for an out-of-bound pointer being returned to callers.

```
static int A[10];

int *test() {
  int *p = A + 10;
  return p; // warn
}

int test(void) {
  int x;
```

```
  return x; // warn: undefined or garbage returned
}
```

alpha.security.cert

SEI CERT checkers which tries to find errors based on their C coding rules.

alpha.security.cert.pos

SEI CERT checkers of POSIX C coding rules.

alpha.security.cert.pos.34c

Finds calls to the `putenv` function which pass a pointer to an automatic variable as the argument.

```
int func(const char *var) {
  char env[1024];
  int retval = snprintf(env, sizeof(env),"TEST=%s", var);
  if (retval < 0 || (size_t)retval >= sizeof(env)) {
      /* Handle error */
  }

  return putenv(env); // putenv function should not be called with auto variables
}
```

Limitations:

- Technically, one can pass automatic variables to `putenv`, but one needs to ensure that the given environment key stays alive until it's removed or overwritten. Since the analyzer cannot keep track of which envvars get overwritten and when, it needs to be slightly more aggressive and warn for such cases too, leading in some cases to false-positive reports like this:

```
void baz() {
  char env[] = "NAME=value";
  putenv(env); // false-positive warning: putenv function should not be called...
  // More code...
  putenv((char *)"NAME=anothervalue");
  // This putenv call overwrites the previous entry, thus that can no longer dangle.
} // 'env' array becomes dead only here.
```

alpha.security.cert.env

SEI CERT checkers of Environment C coding rules.

alpha.security.cert.env.InvalidPtr

Corresponds to SEI CERT Rules ENV31-C and ENV34-C.

ENV31-C: Rule is about the possible problem with *main* function's third argument, environment pointer, "envp". When enviornment array is modified using some modification function such as putenv, setenv or others, It may happen that memory is reallocated, however "envp" is not updated to reflect the changes and points to old memory region.

ENV34-C: Some functions return a pointer to a statically allocated buffer. Consequently, subsequent call of these functions will invalidate previous pointer. These functions include: getenv, localeconv, asctime, setlocale, strerror

```
int main(int argc, const char *argv[], const char *envp[]) {
  if (setenv("MY_NEW_VAR", "new_value", 1) != 0) {
    // setenv call may invalidate 'envp'
    /* Handle error */
  }
  if (envp != NULL) {
    for (size_t i = 0; envp[i] != NULL; ++i) {
      puts(envp[i]);
```

```
      // envp may no longer point to the current environment
      // this program has unanticipated behavior, since envp
      // does not reflect changes made by setenv function.
    }
  }
  return 0;
}

void previous_call_invalidation() {
  char *p, *pp;

  p = getenv("VAR");
  pp = getenv("VAR2");
  // subsequent call to 'getenv' invalidated previous one

  *p;
  // dereferencing invalid pointer
}
```

alpha.security.taint

Checkers implementing taint analysis.

alpha.security.taint.TaintPropagation (C, C++)

Taint analysis identifies untrusted sources of information (taint sources), rules as to how the untrusted data flows along the execution path (propagation rules), and points of execution where the use of tainted data is risky (taints sinks). The most notable examples of taint sources are:

- network originating data
- environment variables
- database originating data

`GenericTaintChecker` is the main implementation checker for this rule, and it generates taint information used by other checkers.

```
void test() {
  char x = getchar(); // 'x' marked as tainted
  system(&x); // warn: untrusted data is passed to a system call
}

// note: compiler internally checks if the second param to
// sprintf is a string literal or not.
// Use -Wno-format-security to suppress compiler warning.
void test() {
  char s[10], buf[10];
  fscanf(stdin, "%s", s); // 's' marked as tainted

  sprintf(buf, s); // warn: untrusted data as a format string
}

void test() {
  size_t ts;
  scanf("%zd", &ts); // 'ts' marked as tainted
  int *p = (int *)malloc(ts * sizeof(int));
    // warn: untrusted data as buffer size
}
```

There are built-in sources, propagations and sinks defined in code inside `GenericTaintChecker`. These operations are handled even if no external taint configuration is provided.

Default sources defined by `GenericTaintChecker`:

`_IO_getc`, `fdopen`, `fopen`, `freopen`, `get_current_dir_name`, `getch`, `getchar`, `getchar_unlocked`, `getwd`, `getcwd`, `getgroups`, `gethostname`, `getlogin`, `getlogin_r`, `getnameinfo`, `gets`, `gets_s`, `getseuserbyname`, `readlink`, `readlinkat`, `scanf`, `scanf_s`, `socket`, `wgetch`

Default propagations defined by `GenericTaintChecker`: `atoi`, `atol`, `atoll`, `basename`, `dirname`, `fgetc`, `fgetln`, `fgets`, `fnmatch`, `fread`, `fscanf`, `fscanf_s`, `index`, `inflate`, `isalnum`, `isalpha`, `isascii`, `isblank`, `iscntrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, `isxdigit`, `memchr`, `memrchr`, `sscanf`, `getc`, `getc_unlocked`, `getdelim`, `getline`, `getw`, `memcmp`, `memcpy`, `memmem`, `memmove`, `mbtowc`, `pread`, `qsort`, `qsort_r`, `rawmemchr`, `read`, `recv`, `recvfrom`, `rindex`, `strcasestr`, `strchr`, `strchrnul`, `strcasecmp`, `strcmp`, `strcspn`, `strlen`, `strncasecmp`, `strncmp`, `strndup`, `strndupa`, `strnlen`, `strpbrk`, `strrchr`, `strsep`, `strspn`, `strstr`, `strtol`, `strtoll`, `strtoul`, `strtoull`, `tolower`, `toupper`, `ttyname`, `ttyname_r`, `wctomb`, `wcwidth`

Default sinks defined in `GenericTaintChecker`: `printf`, `setproctitle`, `system`, `popen`, `execl`, `execle`, `execlp`, `execv`, `execvp`, `execvP`, `execve`, `dlopen`, `memcpy`, `memmove`, `strncpy`, `strndup`, `malloc`, `calloc`, `alloca`, `memccpy`, `realloc`, `bcopy`

The user can configure taint sources, sinks, and propagation rules by providing a configuration file via checker option `alpha.security.taint.TaintPropagation:Config`.

External taint configuration is in YAML format. The taint-related options defined in the config file extend but do not override the built-in sources, rules, sinks. The format of the external taint configuration file is not stable, and could change without any notice even in a non-backward compatible way.

For a more detailed description of configuration options, please see the Taint Analysis Configuration. For an example see Example configuration file.

alpha.unix

alpha.unix.StdCLibraryFunctionArgs (C)

Check for calls of standard library functions that violate predefined argument constraints. For example, it is stated in the C standard that for the `int isalnum(int ch)` function the behavior is undefined if the value of `ch` is not representable as unsigned char and is not equal to `EOF`.

```
void test_alnum_concrete(int v) {
  int ret = isalnum(256); // \
  // warning: Function argument constraint is not satisfied
  (void)ret;
}
```

If the argument's value is unknown then the value is assumed to hold the proper value range.

```
#define EOF -1
int test_alnum_symbolic(int x) {
  int ret = isalnum(x);
  // after the call, ret is assumed to be in the range [-1, 255]

  if (ret > 255)      // impossible (infeasible branch)
    if (x == 0)
      return ret / x; // division by zero is not reported
  return ret;
}
```

If the user disables the checker then the argument violation warning is suppressed. However, the assumption about the argument is still modeled. This is because exploring an execution path that already contains undefined behavior is not valuable.

There are different kind of constraints modeled: range constraint, not null constraint, buffer size constraint. A **range constraint** requires the argument's value to be in a specific range, see `isalnum` as an example above. A **not null constraint** requires the pointer argument to be non-null.

A **buffer size** constraint specifies the minimum size of the buffer argument. The size might be a known constant. For example, `asctime_r` requires that the buffer argument's size must be greater than or equal to 26 bytes. In other

cases, the size is denoted by another argument or as a multiplication of two arguments. For instance, `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)`. Here, `ptr` is the buffer, and its minimum size is `size * nmemb`

```
void buffer_size_constraint_violation(FILE *file) {
  enum { BUFFER_SIZE = 1024 };
  wchar_t wbuf[BUFFER_SIZE];

  const size_t size = sizeof(*wbuf);   // 4
  const size_t nitems = sizeof(wbuf);  // 4096

  // Below we receive a warning because the 3rd parameter should be the
  // number of elements to read, not the size in bytes. This case is a known
  // vulnerability described by the ARR38-C SEI-CERT rule.
  fread(wbuf, size, nitems, file);
}
```

Limitations

The checker is in alpha because the reports cannot provide notes about the values of the arguments. Without this information it is hard to confirm if the constraint is indeed violated. For example, consider the above case for `fread`. We display in the warning message that the size of the 1st arg should be equal to or less than the value of the 2nd arg times the 3rd arg. However, we fail to display the concrete values (`4` and `4096`) for those arguments.

Parameters

The checker models functions (and emits diagnostics) from the C standard by default. The `ModelPOSIX` option enables the checker to model (and emit diagnostics) for functions that are defined in the POSIX standard. This option is disabled by default.

alpha.unix.BlockInCriticalSection (C)

Check for calls to blocking functions inside a critical section. Applies to: `lock, unlock, sleep, getc, fgets, read, recv, pthread_mutex_lock, `` pthread_mutex_unlock, mtx_lock, mtx_timedlock, mtx_trylock, mtx_unlock, lock_guard, unique_lock``

```
void test() {
  std::mutex m;
  m.lock();
  sleep(3); // warn: a blocking function sleep is called inside a critical
            //       section
  m.unlock();
}
```

alpha.unix.Chroot (C)

Check improper use of chroot.

```
void f();

void test() {
  chroot("/usr/local");
  f(); // warn: no call of chdir("/") immediately after chroot
}
```

alpha.unix.PthreadLock (C)

Simple lock -> unlock checker. Applies to: `pthread_mutex_lock, pthread_rwlock_rdlock, pthread_rwlock_wrlock, lck_mtx_lock, lck_rw_lock_exclusive lck_rw_lock_shared, pthread_mutex_trylock, pthread_rwlock_tryrdlock, pthread_rwlock_tryrwlock, lck_mtx_try_lock, lck_rw_try_lock_exclusive, lck_rw_try_lock_shared, pthread_mutex_unlock, pthread_rwlock_unlock, lck_mtx_unlock, lck_rw_done`.

```
pthread_mutex_t mtx;
```

```
void test() {
  pthread_mutex_lock(&mtx);
  pthread_mutex_lock(&mtx);
    // warn: this lock has already been acquired
}

lck_mtx_t lck1, lck2;

void test() {
  lck_mtx_lock(&lck1);
  lck_mtx_lock(&lck2);
  lck_mtx_unlock(&lck1);
    // warn: this was not the most recently acquired lock
}

lck_mtx_t lck1, lck2;

void test() {
  if (lck_mtx_try_lock(&lck1) == 0)
    return;

  lck_mtx_lock(&lck2);
  lck_mtx_unlock(&lck1);
    // warn: this was not the most recently acquired lock
}
```

alpha.unix.SimpleStream (C)

Check for misuses of stream APIs. Check for misuses of stream APIs: `fopen, fclose` (demo checker, the subject of the demo (Slides , Video) by Anna Zaks and Jordan Rose presented at the 2012 LLVM Developers' Meeting).

```
void test() {
  FILE *F = fopen("myfile.txt", "w");
} // warn: opened file is never closed

void test() {
  FILE *F = fopen("myfile.txt", "w");

  if (F)
    fclose(F);

  fclose(F); // warn: closing a previously closed file stream
}
```

alpha.unix.Stream (C)

Check stream handling functions: `fopen, tmpfile, fclose, fread, fwrite, fseek, ftell, rewind, fgetpos, fsetpos, clearerr, feof, ferror, fileno`.

```
void test() {
  FILE *p = fopen("foo", "r");
} // warn: opened file is never closed

void test() {
  FILE *p = fopen("foo", "r");
  fseek(p, 1, SEEK_SET); // warn: stream pointer might be NULL
  fclose(p);
}

void test() {
```

```
  FILE *p = fopen("foo", "r");

  if (p)
    fseek(p, 1, 3);
     // warn: third arg should be SEEK_SET, SEEK_END, or SEEK_CUR

  fclose(p);
}

void test() {
  FILE *p = fopen("foo", "r");
  fclose(p);
  fclose(p); // warn: already closed
}

void test() {
  FILE *p = tmpfile();
  ftell(p); // warn: stream pointer might be NULL
  fclose(p);
}
```

alpha.unix.cstring.BufferOverlap (C)

Checks for overlap in two buffer arguments. Applies to: `memcpy, mempcpy`.

```
void test() {
  int a[4] = {0};
  memcpy(a + 2, a + 1, 8); // warn
}
```

alpha.unix.cstring.NotNullTerminated (C)

Check for arguments which are not null-terminated strings; applies to: `strlen, strnlen, strcpy, strncpy, strcat, strncat`.

```
void test() {
  int y = strlen((char *)&test); // warn
}
```

alpha.unix.cstring.OutOfBounds (C)

Check for out-of-bounds access in string functions; applies to:`` strncopy, strncat``.

```
void test() {
  int y = strlen((char *)&test); // warn
}
```

alpha.unix.cstring.UninitializedRead (C)

Check for uninitialized reads from common memory copy/manipulation functions such as:

`memcpy, mempcpy, memmove, memcmp, strcmp, strncmp, strcpy, strlen, strsep` and many more.

```
void test() {
 char src[10];
 char dst[5];
 memcpy(dst,src,sizeof(dst)); // warn: Bytes string function accesses uninitialized/garbage values
}
```

Limitations:

- Due to limitations of the memory modeling in the analyzer, one can likely observe a lot of false-positive reports like this:

```
void false_positive() {
  int src[] = {1, 2, 3, 4};
  int dst[5] = {0};
  memcpy(dst, src, 4 * sizeof(int)); // false-positive:
  // The 'src' buffer was correctly initialized, yet we cannot conclude
  // that since the analyzer could not see a direct initialization of the
  // very last byte of the source buffer.
}
```

  More details at the corresponding GitHub issue.

alpha.nondeterminism.PointerIteration (C++)

Check for non-determinism caused by iterating unordered containers of pointers.

```
void test() {
 int a = 1, b = 2;
 std::unordered_set<int *> UnorderedPtrSet = {&a, &b};

 for (auto i : UnorderedPtrSet) // warn
   f(i);
}
```

alpha.nondeterminism.PointerSorting (C++)

Check for non-determinism caused by sorting of pointers.

```
void test() {
 int a = 1, b = 2;
 std::vector<int *> V = {&a, &b};
 std::sort(V.begin(), V.end()); // warn
}
```

alpha.WebKit

alpha.webkit.UncountedCallArgsChecker

The goal of this rule is to make sure that lifetime of any dynamically allocated ref-countable object passed as a call argument spans past the end of the call. This applies to call to any function, method, lambda, function pointer or functor. Ref-countable types aren't supposed to be allocated on stack so we check arguments for parameters of raw pointers and references to uncounted types.

Here are some examples of situations that we warn about as they *might* be potentially unsafe. The logic is that either we're able to guarantee that an argument is safe or it's considered if not a bug then bug-prone.

```
RefCountable* provide_uncounted();
void consume(RefCountable*);

// In these cases we can't make sure callee won't directly or indirectly call `deref()` on the argument which could make it unsafe from such point until the end of the call.

void foo1() {
  consume(provide_uncounted()); // warn
}

void foo2() {
  RefCountable* uncounted = provide_uncounted();
  consume(uncounted); // warn
}
```

Although we are enforcing member variables to be ref-counted by *webkit.NoUncountedMemberChecker* any method of the same class still has unrestricted access to these. Since from a caller's perspective we can't guarantee a particular member won't get modified by callee (directly or indirectly) we don't consider values obtained from members safe.

Note: It's likely this heuristic could be made more precise with fewer false positives - for example calls to free functions that don't have any parameter other than the pointer should be safe as the callee won't be able to tamper with the member unless it's a global variable.

```
struct Foo {
  RefPtr<RefCountable> member;
  void consume(RefCountable*) { /* ... */ }
  void bugprone() {
    consume(member.get()); // warn
  }
};
```

The implementation of this rule is a heuristic - we define a whitelist of kinds of values that are considered safe to be passed as arguments. If we can't prove an argument is safe it's considered an error.

Allowed kinds of arguments:

- values obtained from ref-counted objects (including temporaries as those survive the call too)

  ```
  RefCountable* provide_uncounted();
  void consume(RefCountable*);

  void foo() {
    RefPtr<RefCountable> rc = makeRef(provide_uncounted());
    consume(rc.get()); // ok
    consume(makeRef(provide_uncounted()).get()); // ok
  }
  ```

- forwarding uncounted arguments from caller to callee

  ```
  void foo(RefCountable& a) {
    bar(a); // ok
  }
  ```

  Caller of `foo()` is responsible for a's lifetime.

- `this` pointer

  ```
  void Foo::foo() {
    baz(this);  // ok
  }
  ```

  Caller of `foo()` is responsible for keeping the memory pointed to by `this` pointer safe.

- constants

  ```
  foo(nullptr, NULL, 0); // ok
  ```

We also define a set of safe transformations which if passed a safe value as an input provide (usually it's the return value) a safe value (or an object that provides safe values). This is also a heuristic.

- constructors of ref-counted types (including factory methods)
- getters of ref-counted types
- member overloaded operators
- casts
- unary operators like `&` or `*`

alpha.webkit.UncountedLocalVarsChecker

The goal of this rule is to make sure that any uncounted local variable is backed by a ref-counted object with lifetime that is strictly larger than the scope of the uncounted local variable. To be on the safe side we require the scope of an uncounted variable to be embedded in the scope of ref-counted object that backs it.

These are examples of cases that we consider safe:

```
void foo1() {
  RefPtr<RefCountable> counted;
```

```
  // The scope of uncounted is EMBEDDED in the scope of counted.
  {
    RefCountable* uncounted = counted.get(); // ok
  }
}

void foo2(RefPtr<RefCountable> counted_param) {
  RefCountable* uncounted = counted_param.get(); // ok
}

void FooClass::foo_method() {
  RefCountable* uncounted = this; // ok
}
```

Here are some examples of situations that we warn about as they *might* be potentially unsafe. The logic is that either we're able to guarantee that an argument is safe or it's considered if not a bug then bug-prone.

```
void foo1() {
  RefCountable* uncounted = new RefCountable; // warn
}

RefCountable* global_uncounted;
void foo2() {
  RefCountable* uncounted = global_uncounted; // warn
}

void foo3() {
  RefPtr<RefCountable> counted;
  // The scope of uncounted is not EMBEDDED in the scope of counted.
  RefCountable* uncounted = counted.get(); // warn
}
```

We don't warn about these cases - we don't consider them necessarily safe but since they are very common and usually safe we'd introduce a lot of false positives otherwise: - variable defined in condition part of an `if` statement - variable defined in init statement condition of a `for` statement

For the time being we also don't warn about uninitialized uncounted local variables.

Debug Checkers

debug

Checkers used for debugging the analyzer. Debug Checks page contains a detailed description.

debug.AnalysisOrder

Print callbacks that are called during analysis in order.

debug.ConfigDumper

Dump config table.

debug.DumpCFG Display

Control-Flow Graphs.

debug.DumpCallGraph

Display Call Graph.

debug.DumpCalls

Print calls as they are traversed by the engine.

debug.DumpDominators

Print the dominance tree for a given CFG.

debug.DumpLiveVars

Print results of live variable analysis.

debug.DumpTraversal

Print branch conditions as they are traversed by the engine.

debug.ExprInspection

Check the analyzer's understanding of expressions.

debug.Stats

Emit warnings with analyzer statistics.

debug.TaintTest

Mark tainted symbols as such.

debug.ViewCFG

View Control-Flow Graphs using GraphViz.

debug.ViewCallGraph

View Call Graph using GraphViz.

debug.ViewExplodedGraph

View Exploded Graphs using GraphViz.

User Docs

Contents:

Cross Translation Unit (CTU) Analysis

Normally, static analysis works in the boundary of one translation unit (TU). However, with additional steps and configuration we can enable the analysis to inline the definition of a function from another TU.

Overview 587

PCH-based analysis 587

- Manual CTU Analysis 587
- Automated CTU Analysis with CodeChecker 588
- Automated CTU Analysis with scan-build-py (don't do it) 589

On-demand analysis 589

- Manual CTU Analysis 589
- Automated CTU Analysis with CodeChecker 591
- Automated CTU Analysis with scan-build-py (don't do it) 591

Overview

CTU analysis can be used in a variety of ways. The importing of external TU definitions can work with pre-dumped PCH files or generating the necessary AST structure on-demand, during the analysis of the main TU. Driving the static analysis can also be implemented in multiple ways. The most direct way is to specify the necessary commandline options of the Clang frontend manually (and generate the prerequisite dependencies of the specific import method by hand). This process can be automated by other tools, like CodeChecker and scan-build-py (preference for the former).

PCH-based analysis

The analysis needs the PCH dumps of all the translations units used in the project. These can be generated by the Clang Frontend itself, and must be arranged in a specific way in the filesystem. The index, which maps symbols' USR names to PCH dumps containing them must also be generated by the *clang-extdef-mapping*. Entries in the index *must* have an *.ast* suffix if the goal is to use PCH-based analysis, as the lack of that extension signals that the entry is to be used as a source-file, and parsed on-demand. This tool uses a compilation database to determine the compilation flags used. The analysis invocation must be provided with the directory which contains the dumps and the mapping files.

Manual CTU Analysis

Let's consider these source files in our minimal example:

```
// main.cpp
int foo();

int main() {
  return 3 / foo();
}

// foo.cpp
int foo() {
  return 0;
}
```

And a compilation database:

```
[
  {
    "directory": "/path/to/your/project",
    "command": "clang++ -c foo.cpp -o foo.o",
    "file": "foo.cpp"
  },
  {
    "directory": "/path/to/your/project",
    "command": "clang++ -c main.cpp -o main.o",
    "file": "main.cpp"
  }
]
```

We'd like to analyze *main.cpp* and discover the division by zero bug. In order to be able to inline the definition of *foo* from *foo.cpp* first we have to generate the *AST* (or *PCH*) file of *foo.cpp*:

```
$ pwd $ /path/to/your/project
$ clang++ -emit-ast -o foo.cpp.ast foo.cpp
$ # Check that the .ast file is generated:
$ ls
compile_commands.json  foo.cpp.ast  foo.cpp  main.cpp
$
```

The next step is to create a CTU index file which holds the *USR* name and location of external definitions in the source files in format *<USR-Length>:<USR> <File-Path>*:

```
$ clang-extdef-mapping -p . foo.cpp
9:c:@F@foo# /path/to/your/project/foo.cpp
$ clang-extdef-mapping -p . foo.cpp > externalDefMap.txt
```

We have to modify *externalDefMap.txt* to contain the name of the *.ast* files instead of the source files:

```
$ sed -i -e "s/.cpp/.cpp.ast/g" externalDefMap.txt
```

We still have to further modify the *externalDefMap.txt* file to contain relative paths:

```
$ sed -i -e "s|$(pwd)/||g" externalDefMap.txt
```

Now everything is available for the CTU analysis. We have to feed Clang with CTU specific extra arguments:

```
$ pwd
/path/to/your/project
$ clang++ --analyze \
    -Xclang -analyzer-config -Xclang experimental-enable-naive-ctu-analysis=true \
    -Xclang -analyzer-config -Xclang ctu-dir=. \
    -Xclang -analyzer-output=plist-multi-file \
    main.cpp
main.cpp:5:12: warning: Division by zero
  return 3 / foo();
         ~~^~~~~~
1 warning generated.
$ # The plist file with the result is generated.
$ ls -F
compile_commands.json  externalDefMap.txt  foo.ast  foo.cpp  foo.cpp.ast  main.cpp  main.plist
$
```

This manual procedure is error-prone and not scalable, therefore to analyze real projects it is recommended to use *CodeChecker* or *scan-build-py*.

Automated CTU Analysis with CodeChecker

The CodeChecker project fully supports automated CTU analysis with Clang. Once we have set up the *PATH* environment variable and we activated the python *venv* then it is all it takes:

```
$ CodeChecker analyze --ctu compile_commands.json -o reports
$ ls -F
compile_commands.json  foo.cpp  foo.cpp.ast  main.cpp  reports/
$ tree reports
reports
■■■ compile_cmd.json
■■■ compiler_info.json
■■■ foo.cpp_53f6fbf7ab7ec9931301524b551959e2.plist
■■■ main.cpp_23db3d8df52ff0812e6e5a03071c8337.plist
■■■ metadata.json
■■■ unique_compile_commands.json

0 directories, 6 files
$
```

The *plist* files contain the results of the analysis, which may be viewed with the regular analysis tools. E.g. one may use *CodeChecker parse* to view the results in command line:

```
$ CodeChecker parse reports
[HIGH] /home/egbomrt/ctu_mini_raw_project/main.cpp:5:12: Division by zero [core.DivideZero]
  return 3 / foo();
           ^

Found 1 defect(s) in main.cpp


----==== Summary ====----
```

```
----------------------
Filename | Report count
----------------------
main.cpp |            1
----------------------
----------------------
Severity | Report count
----------------------
HIGH     |            1
----------------------
----==================----
Total number of reports: 1
----==================----
```

Or we can use *CodeChecker parse -e html* to export the results into HTML format:

```
$ CodeChecker parse -e html -o html_out reports
$ firefox html_out/index.html
```

Automated CTU Analysis with scan-build-py (don't do it)

We actively develop CTU with CodeChecker as the driver for this feature, *scan-build-py* is not actively developed for CTU. *scan-build-py* has various errors and issues, expect it to work only with the very basic projects only.

Example usage of scan-build-py:

```
$ /your/path/to/llvm-project/clang/tools/scan-build-py/bin/analyze-build --ctu
analyze-build: Run 'scan-view /tmp/scan-build-2019-07-17-17-53-33-810365-7fqgWk' to examine bug reports.
$ /your/path/to/llvm-project/clang/tools/scan-view/bin/scan-view /tmp/scan-build-2019-07-17-17-53-33-810365-7fqgWk
Starting scan-view at: http://127.0.0.1:8181
  Use Ctrl-C to exit.
[6336:6431:0717/175357.633914:ERROR:browser_process_sub_thread.cc(209)] Waited 5 ms for network service
Opening in existing browser session.
^C
$
```

On-demand analysis

The analysis produces the necessary AST structure of external TUs during analysis. This requires the exact compiler invocations for each TU, which can be generated by hand, or by tools driving the analyzer. The compiler invocation is a shell command that could be used to compile the TU-s main source file. The mapping from absolute source file paths of a TU to lists of compilation command segments used to compile said TU are given in YAML format referred to as *invocation list*, and must be passed as an analyer-config argument. The index, which maps function USR names to source files containing them must also be generated by the *clang-extdef-mapping*. Entries in the index must *not* have an *.ast* suffix if the goal is to use On-demand analysis, as that extension signals that the entry is to be used as an PCH-dump. The mapping of external definitions implicitly uses a compilation database to determine the compilation flags used. The analysis invocation must be provided with the directory which contains the mapping files, and the *invocation list* which is used to determine compiler flags.

Manual CTU Analysis

Let's consider these source files in our minimal example:

```
// main.cpp
int foo();

int main() {
  return 3 / foo();
}

// foo.cpp
int foo() {
  return 0;
}
```

The compilation database:

```
[
  {
    "directory": "/path/to/your/project",
    "command": "clang++ -c foo.cpp -o foo.o",
    "file": "foo.cpp"
  },
  {
    "directory": "/path/to/your/project",
    "command": "clang++ -c main.cpp -o main.o",
    "file": "main.cpp"
  }
]
```

The *invocation list*:

```
"/path/to/your/project/foo.cpp":
  - "clang++"
  - "-c"
  - "/path/to/your/project/foo.cpp"
  - "-o"
  - "/path/to/your/project/foo.o"

"/path/to/your/project/main.cpp":
  - "clang++"
  - "-c"
  - "/path/to/your/project/main.cpp"
  - "-o"
  - "/path/to/your/project/main.o"
```

We'd like to analyze *main.cpp* and discover the division by zero bug. As we are using On-demand mode, we only need to create a CTU index file which holds the *USR* name and location of external definitions in the source files in format *<USR-Length>:<USR> <File-Path>*:

```
$ clang-extdef-mapping -p . foo.cpp
9:c:@F@foo# /path/to/your/project/foo.cpp
$ clang-extdef-mapping -p . foo.cpp > externalDefMap.txt
```

Now everything is available for the CTU analysis. We have to feed Clang with CTU specific extra arguments:

```
$ pwd
/path/to/your/project
$ clang++ --analyze \
    -Xclang -analyzer-config -Xclang experimental-enable-naive-ctu-analysis=true \
    -Xclang -analyzer-config -Xclang ctu-dir=. \
    -Xclang -analyzer-config -Xclang ctu-invocation-list=invocations.yaml \
    -Xclang -analyzer-output=plist-multi-file \
    main.cpp
main.cpp:5:12: warning: Division by zero
  return 3 / foo();
         ~~^~~~~~~
1 warning generated.
$ # The plist file with the result is generated.
$ ls -F
compile_commands.json  externalDefMap.txt  foo.cpp  main.cpp  main.plist
$
```

This manual procedure is error-prone and not scalable, therefore to analyze real projects it is recommended to use *CodeChecker* or *scan-build-py*.

Automated CTU Analysis with CodeChecker

The CodeChecker project fully supports automated CTU analysis with Clang. Once we have set up the *PATH* environment variable and we activated the python *venv* then it is all it takes:

```
$ CodeChecker analyze --ctu --ctu-ast-loading-mode on-demand compile_commands.json -o reports
$ ls -F
compile_commands.json  foo.cpp main.cpp  reports/
$ tree reports
reports
■■■ compile_cmd.json
■■■ compiler_info.json
■■■ foo.cpp_53f6fbf7ab7ec9931301524b551959e2.plist
■■■ main.cpp_23db3d8df52ff0812e6e5a03071c8337.plist
■■■ metadata.json
■■■ unique_compile_commands.json

0 directories, 6 files
$
```

The *plist* files contain the results of the analysis, which may be viewed with the regular analysis tools. E.g. one may use *CodeChecker parse* to view the results in command line:

```
$ CodeChecker parse reports
[HIGH] /home/egbomrt/ctu_mini_raw_project/main.cpp:5:12: Division by zero [core.DivideZero]
  return 3 / foo();
           ^

Found 1 defect(s) in main.cpp


----==== Summary ====----
-----------------------
Filename | Report count
-----------------------
main.cpp |            1
-----------------------
-----------------------
Severity | Report count
-----------------------
HIGH     |            1
-----------------------
----==================----
Total number of reports: 1
----==================----
```

Or we can use *CodeChecker parse -e html* to export the results into HTML format:

```
$ CodeChecker parse -e html -o html_out reports
$ firefox html_out/index.html
```

Automated CTU Analysis with scan-build-py (don’t do it)

We actively develop CTU with CodeChecker as the driver for feature, *scan-build-py* is not actively developed for CTU. *scan-build-py* has various errors and issues, expect it to work only with the very basic projects only.

Currently On-demand analysis is not supported with *scan-build-py*.

Taint Analysis Configuration

The Clang Static Analyzer uses taint analysis to detect security-related issues in code. The backbone of taint analysis in the Clang SA is the *GenericTaintChecker*, which the user can access via the alpha.security.taint.TaintPropagation (C, C++) checker alias and this checker has a default taint-related configuration. The built-in default settings are defined in code, and they are always in effect once the checker is enabled, either directly or via the alias. The checker also provides a configuration interface for extending the default settings by providing a configuration file in YAML format. This documentation describes the syntax of the configuration file and gives the informal semantics of the configuration options.

Overview 592

Example configuration file 593

Configuration file syntax and semantics 594

- Filter syntax and semantics 594
- Propagation syntax and semantics 594
- Sink syntax and semantics 595

Overview

Taint analysis works by checking for the occurrence of special operations during the symbolic execution of the program. Taint analysis defines sources, sinks, and propagation rules. It identifies errors by detecting a flow of information that originates from a taint source, reaches a taint sink, and propagates through the program paths via propagation rules. A source, sink, or an operation that propagates taint is mainly domain-specific knowledge, but there are some built-in defaults provided by alpha.security.taint.TaintPropagation (C, C++). It is possible to express that a statement sanitizes tainted values by providing a `Filters` section in the external configuration (see Example configuration file and Filter syntax and semantics). There are no default filters defined in the built-in settings. The checker's documentation also specifies how to provide a custom taint configuration with command-line options.

Example configuration file

```
# The entries that specify arguments use 0-based indexing when specifying
# input arguments, and -1 is used to denote the return value.

Filters:
  # Filter functions
  # Taint is sanitized when tainted variables are pass arguments to filters.

  # Filter function
  #   void cleanse_first_arg(int* arg)
  #
  # Result example:
  #   int x; // x is tainted
  #   cleanse_first_arg(&x); // x is not tainted after the call
  - Name: cleanse_first_arg
    Args: [0]

Propagations:
  # Source functions
  # The omission of SrcArgs key indicates unconditional taint propagation,
  # which is conceptually what a source does.

  # Source function
  #   size_t fread(void *ptr, size_t size, size_t nmemb, FILE * stream)
  #
  # Result example:
  #   FILE* f = fopen("file.txt");
  #   char buf[1024];
  #   size_t read = fread(buf, sizeof(buf[0]), sizeof(buf)/sizeof(buf[0]), f);
  #   // both read and buf are tainted
  - Name: fread
    DstArgs: [0, -1]

  # Propagation functions
  # The presence of SrcArgs key indicates conditional taint propagation,
  # which is conceptually what a propagator does.

  # Propagation function
  #   char *dirname(char *path)
  #
  # Result example:
  #   char* path = read_path();
  #   char* dir = dirname(path);
  #   // dir is tainted if path was tainted
  - Name: dirname
    SrcArgs: [0]
    DstArgs: [-1]

Sinks:
  # Sink functions
  # If taint reaches any of the arguments specified, a warning is emitted.

  # Sink function
  #   int system(const char* command)
  #
  # Result example:
  #   const char* command = read_command();
  #   system(command); // emit diagnostic if command is tainted
```

```
  - Name: system
    Args: [0]
```

In the example file above, the entries under the *Propagation* key implement the conceptual sources and propagations, and sinks have their dedicated *Sinks* key. The user can define operations (function calls) where the tainted values should be cleansed by listing entries under the *Filters* key. Filters model the sanitization of values done by the programmer, and providing these is key to avoiding false-positive findings.

Configuration file syntax and semantics

The configuration file should have valid YAML syntax.

The configuration file can have the following top-level keys:

- Filters
- Propagations
- Sinks

Under the *Filters* key, the user can specify a list of operations that remove taint (see Filter syntax and semantics for details).

Under the *Propagations* key, the user can specify a list of operations that introduce and propagate taint (see Propagation syntax and semantics for details). The user can mark taint sources with a *SrcArgs* key in the *Propagation* key, while propagations have none. The lack of the *SrcArgs* key means unconditional propagation, which is how sources are modeled. The semantics of propagations are such, that if any of the source arguments are tainted (specified by indexes in *SrcArgs*) then all of the destination arguments (specified by indexes in *DstArgs*) also become tainted.

Under the *Sinks* key, the user can specify a list of operations where the checker should emit a bug report if tainted data reaches it (see Sink syntax and semantics for details).

Filter syntax and semantics

An entry under *Filters* is a YAML object with the following mandatory keys:

- *Name* is a string that specifies the name of a function. Encountering this function during symbolic execution the checker will sanitize taint from the memory region referred to by the given arguments or return a sanitized value.
- *Args* is a list of numbers in the range of `[-1..int_max]`. It indicates the indexes of arguments in the function call. The number `-1` signifies the return value; other numbers identify call arguments. The values of these arguments are considered clean after the function call.

The following keys are optional:

- *Scope* is a string that specifies the prefix of the function's name in its fully qualified name. This option restricts the set of matching function calls. It can encode not only namespaces but struct/class names as well to match member functions.

Propagation syntax and semantics

An entry under *Propagation* is a YAML object with the following mandatory keys:

- *Name* is a string that specifies the name of a function. Encountering this function during symbolic execution propagate taint from one or more arguments to other arguments and possibly the return value. It helps model the taint-related behavior of functions that are not analyzable otherwise.

The following keys are optional:

- *Scope* is a string that specifies the prefix of the function's name in its fully qualified name. This option restricts the set of matching function calls.
- *SrcArgs* is a list of numbers in the range of `[0..int_max]` that indicates the indexes of arguments in the function call. Taint-propagation considers the values of these arguments during the evaluation of the

function call. If any *SrcArgs* arguments are tainted, the checker will consider all *DstArgs* arguments tainted after the call.

- *DstArgs* is a list of numbers in the range of `[-1..int_max]` that indicates the indexes of arguments in the function call. The number `-1` specifies the return value of the function. If any *SrcArgs* arguments are tainted, the checker will consider all *DstArgs* arguments tainted after the call.
- *VariadicType* is a string that can be one of `None`, `Dst`, `Src`. It is used in conjunction with *VariadicIndex* to specify arguments inside a variadic argument. The value of `Src` will treat every call site argument that is part of a variadic argument list as a source concerning propagation rules (as if specified by *SrcArg*). The value of `Dst` will treat every call site argument that is part of a variadic argument list a destination concerning propagation rules. The value of `None` will not consider the arguments that are part of a variadic argument list (this option is redundant but can be used to temporarily switch off handling of a particular variadic argument option without removing the VariadicIndex key).
- *VariadicIndex* is a number in the range of `[0..int_max]`. It indicates the starting index of the variadic argument in the signature of the function.

Sink syntax and semantics

An entry under *Sinks* is a YAML object with the following mandatory keys:

- *Name* is a string that specifies the name of a function. Encountering this function during symbolic execution will emit a taint-related diagnostic if any of the arguments specified with *Args* are tainted at the call site.
- *Args* is a list of numbers in the range of `[0..int_max]` that indicates the indexes of arguments in the function call. The checker reports an error if any of the specified arguments are tainted.

The following keys are optional:

- *Scope* is a string that specifies the prefix of the function's name in its fully qualified name. This option restricts the set of matching function calls.

Developer Docs

Contents:

Debug Checks

General Analysis Dumpers 595

Path Tracking 596

State Checking 596

ExprInspection checks 596

Statistics 599

Output testing checkers 599

The analyzer contains a number of checkers which can aid in debugging. Enable them by using the "-analyzer-checker=" flag, followed by the name of the checker.

General Analysis Dumpers

These checkers are used to dump the results of various infrastructural analyses to stderr. Some checkers also have "view" variants, which will display a graph using a 'dot' format viewer (such as Graphviz on macOS) instead.

- debug.DumpCallGraph, debug.ViewCallGraph: Show the call graph generated for the current translation unit. This is used to determine the order in which to analyze functions when inlining is enabled.
- debug.DumpCFG, debug.ViewCFG: Show the CFG generated for each top-level function being analyzed.
- debug.DumpDominators: Shows the dominance tree for the CFG of each top-level function.
- debug.DumpLiveVars: Show the results of live variable analysis for each top-level function being analyzed.
- debug.DumpLiveExprs: Show the results of live expression analysis for each top-level function being analyzed.

- debug.ViewExplodedGraph: Show the Exploded Graphs generated for the analysis of different functions in the input translation unit. When there are several functions analyzed, display one graph per function. Beware that these graphs may grow very large, even for small functions.

Path Tracking

These checkers print information about the path taken by the analyzer engine.

- debug.DumpCalls: Prints out every function or method call encountered during a path traversal. This is indented to show the call stack, but does NOT do any special handling of branches, meaning different paths could end up interleaved.
- debug.DumpTraversal: Prints the name of each branch statement encountered during a path traversal (“IfStmt”, “WhileStmt”, etc). Currently used to check whether the analysis engine is doing BFS or DFS.

State Checking

These checkers will print out information about the analyzer state in the form of analysis warnings. They are intended for use with the -verify functionality in regression tests.

- debug.TaintTest: Prints out the word “tainted” for every expression that carries taint. At the time of this writing, taint was only introduced by the checks under experimental.security.taint.TaintPropagation; this checker may eventually move to the security.taint package.
- debug.ExprInspection: Responds to certain function calls, which are modeled after builtins. These function calls should affect the program state other than the evaluation of their arguments; to use them, you will need to declare them within your test file. The available functions are described below.

(FIXME: debug.ExprInspection should probably be renamed, since it no longer only inspects expressions.)

ExprInspection checks

- `void clang_analyzer_eval(bool);`

  Prints TRUE if the argument is known to have a non-zero value, FALSE if the argument is known to have a zero or null value, and UNKNOWN if the argument isn’t sufficiently constrained on this path. You can use this to test other values by using expressions like “x == 5”. Note that this functionality is currently DISABLED in inlined functions, since different calls to the same inlined function could provide different information, making it difficult to write proper -verify directives.

  In C, the argument can be typed as ‘int’ or as ‘_Bool’.

  Example usage:

  ```
  clang_analyzer_eval(x); // expected-warning{{UNKNOWN}}
  if (!x) return;
  clang_analyzer_eval(x); // expected-warning{{TRUE}}
  ```

- `void clang_analyzer_checkInlined(bool);`

  If a call occurs within an inlined function, prints TRUE or FALSE according to the value of its argument. If a call occurs outside an inlined function, nothing is printed.

  The intended use of this checker is to assert that a function is inlined at least once (by passing ‘true’ and expecting a warning), or to assert that a function is never inlined (by passing ‘false’ and expecting no warning). The argument is technically unnecessary but is intended to clarify intent.

  You might wonder why we can’t print TRUE if a function is ever inlined and FALSE if it is not. The problem is that any inlined function could conceivably also be analyzed as a top-level function (in which case both TRUE and FALSE would be printed), depending on the value of the -analyzer-inlining option.

  In C, the argument can be typed as ‘int’ or as ‘_Bool’.

  Example usage:

  ```
  int inlined() {
    clang_analyzer_checkInlined(true); // expected-warning{{TRUE}}
    return 42;
  ```

```
}

void topLevel() {
  clang_analyzer_checkInlined(false); // no-warning (not inlined)
  int value = inlined();
  // This assertion will not be valid if the previous call was not inlined.
  clang_analyzer_eval(value == 42); // expected-warning{{TRUE}}
}
```

- `void clang_analyzer_warnIfReached();`

  Generate a warning if this line of code gets reached by the analyzer.

  Example usage:

```
if (true) {
  clang_analyzer_warnIfReached();  // expected-warning{{REACHABLE}}
}
else {
  clang_analyzer_warnIfReached();  // no-warning
}
```

- `void clang_analyzer_numTimesReached();`

  Same as above, but include the number of times this call expression gets reached by the analyzer during the current analysis.

  Example usage:

```
for (int x = 0; x < 3; ++x) {
  clang_analyzer_numTimesReached(); // expected-warning{{3}}
}
```

- `void clang_analyzer_warnOnDeadSymbol(int);`

  Subscribe for a delayed warning when the symbol that represents the value of the argument is garbage-collected by the analyzer.

  When calling ‘clang_analyzer_warnOnDeadSymbol(x)’, if value of ‘x’ is a symbol, then this symbol is marked by the ExprInspection checker. Then, during each garbage collection run, the checker sees if the marked symbol is being collected and issues the ‘SYMBOL DEAD’ warning if it does. This way you know where exactly, up to the line of code, the symbol dies.

  It is unlikely that you call this function after the symbol is already dead, because the very reference to it as the function argument prevents it from dying. However, if the argument is not a symbol but a concrete value, no warning would be issued.

  Example usage:

```
do {
  int x = generate_some_integer();
  clang_analyzer_warnOnDeadSymbol(x);
} while(0);  // expected-warning{{SYMBOL DEAD}}
```

- `void clang_analyzer_explain(a single argument of any type);`

  This function explains the value of its argument in a human-readable manner in the warning message. You can make as many overrides of its prototype in the test code as necessary to explain various integral, pointer, or even record-type values. To simplify usage in C code (where overloading the function declaration is not allowed), you may append an arbitrary suffix to the function name, without affecting functionality.

  Example usage:

```
void clang_analyzer_explain(int);
void clang_analyzer_explain(void *);

// Useful in C code
void clang_analyzer_explain_int(int);
```

```
void foo(int param, void *ptr) {
  clang_analyzer_explain(param); // expected-warning{{argument 'param'}}
  clang_analyzer_explain_int(param); // expected-warning{{argument 'param'}}
  if (!ptr)
    clang_analyzer_explain(ptr); // expected-warning{{memory address '0'}}
}
```

- `void clang_analyzer_dump( /* a single argument of any type */);`

  Similar to clang_analyzer_explain, but produces a raw dump of the value, same as SVal::dump().

  Example usage:

```
void clang_analyzer_dump(int);
void foo(int x) {
  clang_analyzer_dump(x); // expected-warning{{reg_$0<x>}}
}
```

- `size_t clang_analyzer_getExtent(void *);`

  This function returns the value that represents the extent of a memory region pointed to by the argument. This value is often difficult to obtain otherwise, because no valid code that produces this value. However, it may be useful for testing purposes, to see how well does the analyzer model region extents.

  Example usage:

```
void foo() {
  int x, *y;
  size_t xs = clang_analyzer_getExtent(&x);
  clang_analyzer_explain(xs); // expected-warning{{'4'}}
  size_t ys = clang_analyzer_getExtent(&y);
  clang_analyzer_explain(ys); // expected-warning{{'8'}}
}
```

- `void clang_analyzer_printState();`

  Dumps the current ProgramState to the stderr. Quickly lookup the program state at any execution point without ViewExplodedGraph or re-compiling the program. This is not very useful for writing tests (apart from testing how ProgramState gets printed), but useful for debugging tests. Also, this method doesn't produce a warning, so it gets printed on the console before all other ExprInspection warnings.

  Example usage:

```
void foo() {
  int x = 1;
  clang_analyzer_printState(); // Read the stderr!
}
```

- `void clang_analyzer_hashDump(int);`

  The analyzer can generate a hash to identify reports. To debug what information is used to calculate this hash it is possible to dump the hashed string as a warning of an arbitrary expression using the function above.

  Example usage:

```
void foo() {
  int x = 1;
  clang_analyzer_hashDump(x); // expected-warning{{hashed string for x}}
}
```

- `void clang_analyzer_denote(int, const char *);`

  Denotes symbols with strings. A subsequent call to clang_analyzer_express() will expresses another symbol in terms of these string. Useful for testing relationships between different symbols.

  Example usage:

```
void foo(int x) {
  clang_analyzer_denote(x, "$x");
```

```
  clang_analyzer_express(x + 1); // expected-warning{{$x + 1}}
}
```

- `void clang_analyzer_express(int);`

  See clang_analyzer_denote().

- `void clang_analyzer_isTainted(a single argument of any type);`

  Queries the analyzer whether the expression used as argument is tainted or not. This is useful in tests, where we don't want to issue warning for all tainted expressions but only check for certain expressions. This would help to reduce the *noise* that the *TaintTest* debug checker would introduce and let you focus on the *expected-warning*'s that you really care about.

  Example usage:

```
int read_integer() {
  int n;
  clang_analyzer_isTainted(n);     // expected-warning{{NO}}
  scanf("%d", &n);
  clang_analyzer_isTainted(n);     // expected-warning{{YES}}
  clang_analyzer_isTainted(n + 2); // expected-warning{{YES}}
  clang_analyzer_isTainted(n > 0); // expected-warning{{YES}}
  int next_tainted_value = n; // no-warning
  return n;
}
```

- `clang_analyzer_dumpExtent(a single argument of any type)`
- `clang_analyzer_dumpElementCount(a single argument of any type)`

  Dumps out the extent and the element count of the argument.

  Example usage:

```
void array() {
  int a[] = {1, 3};
  clang_analyzer_dumpExtent(a);       // expected-warning {{8 S64b}}
  clang_analyzer_dumpElementCount(a); // expected-warning {{2 S64b}}
}
```

Statistics

The debug.Stats checker collects various information about the analysis of each function, such as how many blocks were reached and if the analyzer timed out.

There is also an additional -analyzer-stats flag, which enables various statistics within the analyzer engine. Note the Stats checker (which produces at least one bug report per function) may actually change the values reported by -analyzer-stats.

Output testing checkers

- debug.ReportStmts reports a warning at **every** statement, making it a very useful tool for testing thoroughly bug report construction and output emission.

Inlining

There are several options that control which calls the analyzer will consider for inlining. The major one is `-analyzer-config ipa`:

- `analyzer-config ipa=none` - All inlining is disabled. This is the only mode available in LLVM 3.1 and earlier and in Xcode 4.3 and earlier.
- `analyzer-config ipa=basic-inlining` **- Turns on inlining for C functions, C++**

  static member functions, and blocks – essentially, the calls that behave like simple C function calls. This is essentially the mode used in Xcode 4.4.

- `analyzer-config ipa=inlining` **- Turns on inlining when we can confidently find**

  the function/method body corresponding to the call. (C functions, static functions, devirtualized C++ methods, Objective-C class methods, Objective-C instance methods when ExprEngine is confident about the dynamic type of the instance).

- `analyzer-config ipa=dynamic` **- Inline instance methods for which the type is**

  determined at runtime and we are not 100% sure that our type info is correct. For virtual calls, inline the most plausible definition.

- `analyzer-config ipa=dynamic-bifurcate` **- Same as -analyzer-config ipa=dynamic,**

  but the path is split. We inline on one branch and do not inline on the other. This mode does not drop the coverage in cases when the parent class has code that is only exercised when some of its methods are overridden.

Currently, `-analyzer-config ipa=dynamic-bifurcate` is the default mode.

While `-analyzer-config ipa` determines in general how aggressively the analyzer will try to inline functions, several additional options control which types of functions can inlined, in an all-or-nothing way. These options use the analyzer's configuration table, so they are all specified as follows:

```
-analyzer-config OPTION=VALUE
```

c++-inlining

This option controls which C++ member functions may be inlined.

```
-analyzer-config c++-inlining=[none | methods | constructors | destructors]
```

Each of these modes implies that all the previous member function kinds will be inlined as well; it doesn't make sense to inline destructors without inlining constructors, for example.

The default c++-inlining mode is 'destructors', meaning that all member functions with visible definitions will be considered for inlining. In some cases the analyzer may still choose not to inline the function.

Note that under 'constructors', constructors for types with non-trivial destructors will not be inlined. Additionally, no C++ member functions will be inlined under -analyzer-config ipa=none or -analyzer-config ipa=basic-inlining, regardless of the setting of the c++-inlining mode.

c++-template-inlining

This option controls whether C++ templated functions may be inlined.

```
-analyzer-config c++-template-inlining=[true | false]
```

Currently, template functions are considered for inlining by default.

The motivation behind this option is that very generic code can be a source of false positives, either by considering paths that the caller considers impossible (by some unstated precondition), or by inlining some but not all of a deep implementation of a function.

c++-stdlib-inlining

This option controls whether functions from the C++ standard library, including methods of the container classes in the Standard Template Library, should be considered for inlining.

```
-analyzer-config c++-stdlib-inlining=[true | false]
```

Currently, C++ standard library functions are considered for inlining by default.

The standard library functions and the STL in particular are used ubiquitously enough that our tolerance for false positives is even lower here. A false positive due to poor modeling of the STL leads to a poor user experience, since most users would not be comfortable adding assertions to system headers in order to silence analyzer warnings.

c++-container-inlining

This option controls whether constructors and destructors of "container" types should be considered for inlining.

```
-analyzer-config c++-container-inlining=[true | false]
```

Currently, these constructors and destructors are NOT considered for inlining by default.

The current implementation of this setting checks whether a type has a member named ‘iterator’ or a member named ‘begin’; these names are idiomatic in C++, with the latter specified in the C++11 standard. The analyzer currently does a fairly poor job of modeling certain data structure invariants of container-like objects. For example, these three expressions should be equivalent:

```
std::distance(c.begin(), c.end()) == 0
c.begin() == c.end()
c.empty()
```

Many of these issues are avoided if containers always have unknown, symbolic state, which is what happens when their constructors are treated as opaque. In the future, we may decide specific containers are “safe” to model through inlining, or choose to model them directly using checkers instead.

Basics of Implementation

The low-level mechanism of inlining a function is handled in ExprEngine::inlineCall and ExprEngine::processCallExit.

If the conditions are right for inlining, a CallEnter node is created and added to the analysis work list. The CallEnter node marks the change to a new LocationContext representing the called function, and its state includes the contents of the new stack frame. When the CallEnter node is actually processed, its single successor will be an edge to the first CFG block in the function.

Exiting an inlined function is a bit more work, fortunately broken up into reasonable steps:

1. The CoreEngine realizes we’re at the end of an inlined call and generates a CallExitBegin node.
2. ExprEngine takes over (in processCallExit) and finds the return value of the function, if it has one. This is bound to the expression that triggered the call. (In the case of calls without origin expressions, such as destructors, this step is skipped.)
3. Dead symbols and bindings are cleaned out from the state, including any local bindings.
4. A CallExitEnd node is generated, which marks the transition back to the caller’s LocationContext.
5. Custom post-call checks are processed and the final nodes are pushed back onto the work list, so that evaluation of the caller can continue.

Retry Without Inlining

In some cases, we would like to retry analysis without inlining a particular call.

Currently, we use this technique to recover coverage in case we stop analyzing a path due to exceeding the maximum block count inside an inlined function.

When this situation is detected, we walk up the path to find the first node before inlining was started and enqueue it on the WorkList with a special ReplayWithoutInlining bit added to it (ExprEngine::replayWithoutInlining). The path is then re-analyzed from that point without inlining that particular call.

Deciding When to Inline

In general, the analyzer attempts to inline as much as possible, since it provides a better summary of what actually happens in the program. There are some cases, however, where the analyzer chooses not to inline:

- If there is no definition available for the called function or method. In this case, there is no opportunity to inline.
- If the CFG cannot be constructed for a called function, or the liveness cannot be computed. These are prerequisites for analyzing a function body, with or without inlining.
- If the LocationContext chain for a given ExplodedNode reaches a maximum cutoff depth. This prevents unbounded analysis due to infinite recursion, but also serves as a useful cutoff for performance reasons.
- If the function is variadic. This is not a hard limitation, but an engineering limitation.

  Tracked by: <rdar://problem/12147064> Support inlining of variadic functions
- In C++, constructors are not inlined unless the destructor call will be processed by the ExprEngine. Thus, if the CFG was built without nodes for implicit destructors, or if the destructors for the given object are not

represented in the CFG, the constructor will not be inlined. (As an exception, constructors for objects with trivial constructors can still be inlined.) See “C++ Caveats” below.

- In C++, ExprEngine does not inline custom implementations of operator ‘new’ or operator ‘delete’, nor does it inline the constructors and destructors associated with these. See “C++ Caveats” below.
- Calls resulting in “dynamic dispatch” are specially handled. See more below.
- The FunctionSummaries map stores additional information about declarations, some of which is collected at runtime based on previous analyses. We do not inline functions which were not profitable to inline in a different context (for example, if the maximum block count was exceeded; see “Retry Without Inlining”).

Dynamic Calls and Devirtualization

“Dynamic” calls are those that are resolved at runtime, such as C++ virtual method calls and Objective-C message sends. Due to the path-sensitive nature of the analysis, the analyzer may be able to reason about the dynamic type of the object whose method is being called and thus “devirtualize” the call.

This path-sensitive devirtualization occurs when the analyzer can determine what method would actually be called at runtime. This is possible when the type information is constrained enough for a simulated C++/Objective-C object that the analyzer can make such a decision.

DynamicTypeInfo

As the analyzer analyzes a path, it may accrue information to refine the knowledge about the type of an object. This can then be used to make better decisions about the target method of a call.

Such type information is tracked as DynamicTypeInfo. This is path-sensitive data that is stored in ProgramState, which defines a mapping from MemRegions to an (optional) DynamicTypeInfo.

If no DynamicTypeInfo has been explicitly set for a MemRegion, it will be lazily inferred from the region’s type or associated symbol. Information from symbolic regions is weaker than from true typed regions.

EXAMPLE: A C++ object declared “A obj” is known to have the class ‘A’, but a

reference “A &ref” may dynamically be a subclass of ‘A’.

The DynamicTypePropagation checker gathers and propagates DynamicTypeInfo, updating it as information is observed along a path that can refine that type information for a region.

WARNING: Not all of the existing analyzer code has been retrofitted to use

DynamicTypeInfo, nor is it universally appropriate. In particular, DynamicTypeInfo always applies to a region with all casts stripped off, but sometimes the information provided by casts can be useful.

RuntimeDefinition

The basis of devirtualization is CallEvent’s getRuntimeDefinition() method, which returns a RuntimeDefinition object. When asked to provide a definition, the CallEvents for dynamic calls will use the DynamicTypeInfo in their ProgramState to attempt to devirtualize the call. In the case of no dynamic dispatch, or perfectly constrained devirtualization, the resulting RuntimeDefinition contains a Decl corresponding to the definition of the called function, and RuntimeDefinition::mayHaveOtherDefinitions will return FALSE.

In the case of dynamic dispatch where our information is not perfect, CallEvent can make a guess, but RuntimeDefinition::mayHaveOtherDefinitions will return TRUE. The RuntimeDefinition object will then also include a MemRegion corresponding to the object being called (i.e., the “receiver” in Objective-C parlance), which ExprEngine uses to decide whether or not the call should be inlined.

Inlining Dynamic Calls

The -analyzer-config ipa option has five different modes: none, basic-inlining, inlining, dynamic, and dynamic-bifurcate. Under -analyzer-config ipa=dynamic, all dynamic calls are inlined, whether we are certain or not that this will actually be the definition used at runtime. Under -analyzer-config ipa=inlining, only “near-perfect” devirtualized calls are inlined*, and other dynamic calls are evaluated conservatively (as if no definition were available).

- Currently, no Objective-C messages are not inlined under -analyzer-config ipa=inlining, even if we are reasonably confident of the type of the receiver. We plan to enable this once we have tested our heuristics more thoroughly.

The last option, -analyzer-config ipa=dynamic-bifurcate, behaves similarly to “dynamic”, but performs a conservative invalidation in the general virtual case in *addition* to inlining. The details of this are discussed below.

As stated above, -analyzer-config ipa=basic-inlining does not inline any C++ member functions or Objective-C method calls, even if they are non-virtual or can be safely devirtualized.

Bifurcation

ExprEngine::BifurcateCall implements the `-analyzer-config ipa=dynamic-bifurcate` mode.

When a call is made on an object with imprecise dynamic type information (RuntimeDefinition::mayHaveOtherDefinitions() evaluates to TRUE), ExprEngine bifurcates the path and marks the object’s region (retrieved from the RuntimeDefinition object) with a path-sensitive “mode” in the ProgramState.

Currently, there are 2 modes:

- `DynamicDispatchModeInlined` **- Models the case where the dynamic type information**

  of the receiver (MemoryRegion) is assumed to be perfectly constrained so that a given definition of a method is expected to be the code actually called. When this mode is set, ExprEngine uses the Decl from RuntimeDefinition to inline any dynamically dispatched call sent to this receiver because the function definition is considered to be fully resolved.

- `DynamicDispatchModeConservative` **- Models the case where the dynamic type**

  information is assumed to be incorrect, for example, implies that the method definition is overridden in a subclass. In such cases, ExprEngine does not inline the methods sent to the receiver (MemoryRegion), even if a candidate definition is available. This mode is conservative about simulating the effects of a call.

Going forward along the symbolic execution path, ExprEngine consults the mode of the receiver’s MemRegion to make decisions on whether the calls should be inlined or not, which ensures that there is at most one split per region.

At a high level, “bifurcation mode” allows for increased semantic coverage in cases where the parent method contains code which is only executed when the class is subclassed. The disadvantages of this mode are a (considerable?) performance hit and the possibility of false positives on the path where the conservative mode is used.

Objective-C Message Heuristics

ExprEngine relies on a set of heuristics to partition the set of Objective-C method calls into those that require bifurcation and those that do not. Below are the cases when the DynamicTypeInfo of the object is considered precise (cannot be a subclass):

- If the object was created with +alloc or +new and initialized with an -init method.
- If the calls are property accesses using dot syntax. This is based on the assumption that children rarely override properties, or do so in an essentially compatible way.
- If the class interface is declared inside the main source file. In this case it is unlikely that it will be subclassed.
- If the method is not declared outside of main source file, either by the receiver’s class or by any superclasses.

C++ Caveats

C++11 [class.cdtor]p4 describes how the vtable of an object is modified as it is being constructed or destructed; that is, the type of the object depends on which base constructors have been completed. This is tracked using DynamicTypeInfo in the DynamicTypePropagation checker.

There are several limitations in the current implementation:

- Temporaries are poorly modeled right now because we’re not confident in the placement of their destructors in the CFG. We currently won’t inline their constructors unless the destructor is trivial, and don’t process their destructors at all, not even to invalidate the region.

- ‘new’ is poorly modeled due to some nasty CFG/design issues. This is tracked in PR12014. ‘delete’ is not modeled at all.
- Arrays of objects are modeled very poorly right now. ExprEngine currently only simulates the first constructor and first destructor. Because of this, ExprEngine does not inline any constructors or destructors for arrays.

CallEvent

A CallEvent represents a specific call to a function, method, or other body of code. It is path-sensitive, containing both the current state (ProgramStateRef) and stack space (LocationContext), and provides uniform access to the argument values and return type of a call, no matter how the call is written in the source or what sort of code body is being invoked.

> **NOTE: For those familiar with Cocoa, CallEvent is roughly equivalent to**
>
> NSInvocation.

CallEvent should be used whenever there is logic dealing with function calls that does not care how the call occurred.

Examples include checking that arguments satisfy preconditions (such as __attribute__((nonnull))), and attempting to inline a call.

CallEvents are reference-counted objects managed by a CallEventManager. While there is no inherent issue with persisting them (say, in a ProgramState’s GDM), they are intended for short-lived use, and can be recreated from CFGElements or non-top-level StackFrameContexts fairly easily.

Initializer List

This discussion took place in https://reviews.llvm.org/D35216 “Escape symbols when creating std::initializer_list”.

It touches problems of modelling C++ standard library constructs in general, including modelling implementation-defined fields within C++ standard library objects, in particular constructing objects into pointers held by such fields, and separation of responsibilities between analyzer’s core and checkers.

Artem:

I’ve seen a few false positives that appear because we construct C++11 std::initializer_list objects with brace initializers, and such construction is not properly modeled. For instance, if a new object is constructed on the heap only to be put into a brace-initialized STL container, the object is reported to be leaked.

Approach (0): This can be trivially fixed by this patch, which causes pointers passed into initializer list expressions to immediately escape.

This fix is overly conservative though. So i did a bit of investigation as to how model std::initializer_list better.

According to the standard, `std::initializer_list<T>` is an object that has methods `begin(), end(), and size()`, where `begin()` returns a pointer to continuous array of `size()` objects of type T, and end() is equal to begin() plus size(). The standard does hint that it should be possible to implement `std::initializer_list<T>` as a pair of pointers, or as a pointer and a size integer, however specific fields that the object would contain are an implementation detail.

Ideally, we should be able to model the initializer list’s methods precisely. Or, at least, it should be possible to explain to the analyzer that the list somehow “takes hold” of the values put into it. Initializer lists can also be copied, which is a separate story that i’m not trying to address here.

The obvious approach to modeling `std::initializer_list` in a checker would be to construct a SymbolMetadata for the memory region of the initializer list object, which would be of type `T*` and represent `begin()`, so we’d trivially model `begin()` as a function that returns this symbol. The array pointed to by that symbol would be `bindLoc()``ed to contain the list's contents (probably as a ``CompoundVal` to produce less bindings in the store). Extent of this array would represent `size()` and would be equal to the length of the list as written.

So this sounds good, however apparently it does nothing to address our false positives: when the list escapes, our `RegionStoreManager` is not magically guessing that the metadata symbol attached to it, together with its contents, should also escape. In fact, it’s impossible to trigger a pointer escape from within the checker.

Approach (1): If only we enabled `ProgramState::bindLoc(..., notifyChanges=true)` to cause pointer escapes (not only region changes) (which sounds like the right thing to do anyway) such checker would be able to solve the false positives by triggering escapes when binding list elements to the list. However, it’d be as conservative

as the current patch's solution. Ideally, we do not want escapes to happen so early. Instead, we'd prefer them to be delayed until the list itself escapes.

So i believe that escaping metadata symbols whenever their base regions escape would be the right thing to do. Currently we didn't think about that because we had neither pointer-type metadatas nor non-pointer escapes.

Approach (2): We could teach the Store to scan itself for bindings to metadata-symbolic-based regions during scanReachableSymbols() whenever a region turns out to be reachable. This requires no work on checker side, but it sounds performance-heavy.

Approach (3): We could let checkers maintain the set of active metadata symbols in the program state (ideally somewhere in the Store, which sounds weird but causes the smallest amount of layering violations), so that the core knew what to escape. This puts a stress on the checkers, but with a smart data map it wouldn't be a problem.

Approach (4): We could allow checkers to trigger pointer escapes in arbitrary moments. If we allow doing this within `checkPointerEscape` callback itself, we would be able to express facts like "when this region escapes, that metadata symbol attached to it should also escape". This sounds like an ultimate freedom, with maximum stress on the checkers - still not too much stress when we have smart data maps.

I'm personally liking the approach (2) - it should be possible to avoid performance overhead, and clarity seems nice.

Gabor:

At this point, I am a bit wondering about two questions.

- When should something belong to a checker and when should something belong to the engine? Sometimes we model library aspects in the engine and model language constructs in checkers.
- What is the checker programming model that we are aiming for? Maximum freedom or more easy checker development?

I think if we aim for maximum freedom, we do not need to worry about the potential stress on checkers, and we can introduce abstractions to mitigate that later on. If we want to simplify the API, then maybe it makes more sense to move language construct modeling to the engine when the checker API is not sufficient instead of complicating the API.

Right now I have no preference or objections between the alternatives but there are some random thoughts:

- Maybe it would be great to have a guideline how to evolve the analyzer and follow it, so it can help us to decide in similar situations
- I do care about performance in this case. The reason is that we have a limited performance budget. And I think we should not expect most of the checker writers to add modeling of language constructs. So, in my opinion, it is ok to have less nice/more verbose API for language modeling if we can have better performance this way, since it only needs to be done once, and is done by the framework developers.

Artem: These are some great questions, i guess it'd be better to discuss them more openly. As a quick dump of my current mood:

- To me it seems obvious that we need to aim for a checker API that is both simple and powerful. This can probably by keeping the API as powerful as necessary while providing a layer of simple ready-made solutions on top of it. Probably a few reusable components for assembling checkers. And this layer should ideally be pleasant enough to work with, so that people would prefer to extend it when something is lacking, instead of falling back to the complex omnipotent API. I'm thinking of AST matchers vs. AST visitors as a roughly similar situation: matchers are not omnipotent, but they're so nice.
- Separation between core and checkers is usually quite strange. Once we have shared state traits, i generally wouldn't mind having region store or range constraint manager as checkers (though it's probably not worth it to transform them - just a mood). The main thing to avoid here would be the situation when the checker overwrites stuff written by the core because it thinks it has a better idea what's going on, so the core should provide a good default behavior.
- Yeah, i totally care about performance as well, and if i try to implement approach, i'd make sure it's good.

Artem:

> Approach (2): We could teach the Store to scan itself for bindings to > metadata-symbolic-based regions during scanReachableSymbols() whenever > a region turns out to be reachable. This requires no work on checker side, > but it sounds performance-heavy.

Nope, this approach is wrong. Metadata symbols may become out-of-date: when the object changes, metadata symbols attached to it aren't changing (because symbols simply don't change). The same metadata may have different symbols to denote its value in different moments of time, but at most one of them represents the actual metadata value. So we'd be escaping more stuff than necessary.

If only we had "ghost fields" (https://lists.llvm.org/pipermail/cfe-dev/2016-May/049000.html), it would have been much easier, because the ghost field would only contain the actual metadata, and the Store would always know about it. This example adds to my belief that ghost fields are exactly what we need for most C++ checkers.

Devin:

In this case, I would be fine with some sort of AbstractStorageMemoryRegion that meant "here is a memory region and somewhere reachable from here exists another region of type T". Or even multiple regions with different identifiers. This wouldn't specify how the memory is reachable, but it would allow for transfer functions to get at those regions and it would allow for invalidation.

For `std::initializer_list` this reachable region would the region for the backing array and the transfer functions for begin() and end() yield the beginning and end element regions for it.

In my view this differs from ghost variables in that (1) this storage does actually exist (it is just a library implementation detail where that storage lives) and (2) it is perfectly valid for a pointer into that storage to be returned and for another part of the program to read or write from that storage. (Well, in this case just read since it is allowed to be read-only memory).

What I'm not OK with is modeling abstract analysis state (for example, the count of a NSMutableArray or the typestate of a file handle) as a value stored in some ginned up region in the store. This takes an easy problem that the analyzer does well at (modeling typestate) and turns it into a hard one that the analyzer is bad at (reasoning about the contents of the heap).

I think the key criterion here is: "is the region accessible from outside the library". That is, does the library expose the region as a pointer that can be read to or written from in the client program? If so, then it makes sense for this to be in the store: we are modeling reachable storage as storage. But if we're just modeling arbitrary analysis facts that need to be invalidated when a pointer escapes then we shouldn't try to gin up storage for them just to get invalidation for free.

Artem:

> In this case, I would be fine with some sort of `AbstractStorageMemoryRegion` > that meant "here is a memory region and somewhere reachable from here exists > another region of type T". Or even multiple regions with different > identifiers. This wouldn't specify how the memory is reachable, but it would > allow for transfer functions to get at those regions and it would allow for > invalidation.

Yeah, this is what we can easily implement now as a symbolic-region-based-on-a-metadata-symbol (though we can make a new region class for that if we eg. want it typed). The problem is that the relation between such storage region and its parent object region is essentially immaterial, similarly to the relation between `SymbolRegionValue` and its parent region. Region contents are mutable: today the abstract storage is reachable from its parent object, tomorrow it's not, and maybe something else becomes reachable, something that isn't even abstract. So the parent region for the abstract storage is most of the time at best a "nice to know" thing - we cannot rely on it to do any actual work. We'd anyway need to rely on the checker to do the job.

> For std::initializer_list this reachable region would the region for the > backing array and the transfer functions for begin() and end() yield the > beginning and end element regions for it.

So maybe in fact for std::initializer_list it may work fine because you cannot change the data after the object is constructed - so this region's contents are essentially immutable. For the future, i feel as if it is a dead end.

I'd like to consider another funny example. Suppose we're trying to model

```
std::unique_ptr. Consider::

  void bar(const std::unique_ptr<int> &x);

  void foo(std::unique_ptr<int> &x) {
    int *a = x.get();   // (a, 0, direct): &AbstractStorageRegion
    *a = 1;             // (AbstractStorageRegion, 0, direct): 1 S32b
    int *b = new int;
    *b = 2;             // (SymRegion{conj_$0<int *>}, 0 ,direct): 2 S32b
```

```
  x.reset(b);         // Checker map: x -> SymRegion{conj_$0<int *>}
  bar(x);             // 'a' doesn't escape (the pointer was unique), 'b' does.
  clang_analyzer_eval(*a == 1); // Making this true is up to the checker.
  clang_analyzer_eval(*b == 2); // Making this unknown is up to the checker.
}
```

The checker doesn't totally need to ensure that `*a == 1` passes - even though the pointer was unique, it could theoretically have `.get()`-ed above and the code could of course break the uniqueness invariant (though we'd probably want it). The checker can say that "even if `*a` did escape, it was not because it was stuffed directly into bar()".

The checker's direct responsibility, however, is to solve the `*b == 2` thing (which is in fact the problem we're dealing with in this patch - escaping the storage region of the object).

So we're talking about one more operation over the program state (scanning reachable symbols and regions) that cannot work without checker support.

We can probably add a new callback "checkReachableSymbols" to solve this. This is in fact also related to the dead symbols problem (we're scanning for live symbols in the store and in the checkers separately, but we need to do so simultaneously with a single worklist). Hmm, in fact this sounds like a good idea; we can replace checkLiveSymbols with checkReachableSymbols.

Or we could just have ghost member variables, and no checker support required at all. For ghost member variables, the relation with their parent region (which would be their superregion) is actually useful, the mutability of their contents is expressed naturally, and the store automagically sees reachable symbols, live symbols, escapes, invalidations, whatever.

> In my view this differs from ghost variables in that (1) this storage does > actually exist (it is just a library implementation detail where that storage > lives) and (2) it is perfectly valid for a pointer into that storage to be > returned and for another part of the program to read or write from that > storage. (Well, in this case just read since it is allowed to be read-only > memory).

> What I'm not OK with is modeling abstract analysis state (for example, the > count of a NSMutableArray or the typestate of a file handle) as a value stored > in some ginned up region in the store.This takes an easy problem that the > analyzer does well at (modeling typestate) and turns it into a hard one that > the analyzer is bad at (reasoning about the contents of the heap).

Yeah, i tend to agree on that. For simple typestates, this is probably an overkill, so let's definitely put aside the idea of "ghost symbolic regions" that i had earlier.

But, to summarize a bit, in our current case, however, the typestate we're looking for is the contents of the heap. And when we try to model such typestates (complex in this specific manner, i.e. heap-like) in any checker, we have a choice between re-doing this modeling in every such checker (which is something analyzer is indeed good at, but at a price of making checkers heavy) or instead relying on the Store to do exactly what it's designed to do.

> I think the key criterion here is: "is the region accessible from outside > the library". That is, does the library expose the region as a pointer that > can be read to or written from in the client program? If so, then it makes > sense for this to be in the store: we are modeling reachable storage as > storage. But if we're just modeling arbitrary analysis facts that need to be > invalidated when a pointer escapes then we shouldn't try to gin up storage > for them just to get invalidation for free.

As a metaphor, i'd probably compare it to body farms - the difference between ghost member variables and metadata symbols seems to me like the difference between body farms and evalCall. Both are nice to have, and body farms are very pleasant to work with, even if not omnipotent. I think it's fine for a FunctionDecl's body in a body farm to have a local variable, even if such variable doesn't actually exist, even if it cannot be seen from outside the function call. I'm not seeing immediate practical difference between "it does actually exist" and "it doesn't actually exist, just a handy abstraction". Similarly, i think it's fine if we have a `CXXRecordDecl` with implementation-defined contents, and try to farm up a member variable as a handy abstraction (we don't even need to know its name or offset, only that it's there somewhere).

Artem:

We've discussed it in person with Devin, and he provided more points to think about:

- If the initializer list consists of non-POD data, constructors of list's objects need to take the sub-region of the list's region as this-region In the current (v2) version of this patch, these objects are constructed elsewhere and

then trivial-copied into the list's metadata pointer region, which may be incorrect. This is our overall problem with C++ constructors, which manifests in this case as well. Additionally, objects would need to be constructed in the analyzer's core, which would not be able to predict that it needs to take a checker-specific region as this-region, which makes it harder, though it might be mitigated by sharing the checker state traits.

- Because "ghost variables" are not material to the user, we need to somehow make super sure that they don't make it into the diagnostic messages.

So, because this needs further digging into overall C++ support and rises too many questions, i'm delaying a better approach to this problem and will fall back to the original trivial patch.

Nullability Checks

This document is a high level description of the nullablility checks. These checks intended to use the annotations that is described in this RFC: http://lists.cs.uiuc.edu/pipermail/cfe-dev/2015-March/041798.html.

Let's consider the following 2 categories:

1) nullable

If a pointer p has a nullable annotation and no explicit null check or assert, we should warn in the following cases:

- p gets implicitly converted into nonnull pointer, for example, we are passing it to a function that takes a nonnull parameter.
- p gets dereferenced

Taking a branch on nullable pointers are the same like taking branch on null unspecified pointers.

Explicit cast from nullable to nonnul:

```
__nullable id foo;
id bar = foo;
takesNonNull((_nonnull) bar); // should not warn here (backward compatibility hack)
anotherTakesNonNull(bar); // would be great to warn here, but not necessary(*)
```

Because bar corresponds to the same symbol all the time it is not easy to implement the checker that way the cast only suppress the first call but not the second. For this reason in the first implementation after a contradictory cast happens, I will treat bar as nullable unspecified, this way all of the warnings will be suppressed. Treating the symbol as nullable unspecified also has an advantage that in case the takesNonNull function body is being inlined, the will be no warning, when the symbol is dereferenced. In case I have time after the initial version I might spend additional time to try to find a more sophisticated solution, in which we would produce the second warning (*).

2) nonnull

- Dereferencing a nonnull, or sending message to it is ok.
- Converting nonnull to nullable is Ok.
- When there is an explicit cast from nonnull to nullable I will trust the cast (it is probable there for a reason, because this cast does not suppress any warnings or errors).
- But what should we do about null checks?:

```
__nonnull id takesNonnull(__nonnull id x) {
    if (x == nil) {
        // Defensive backward compatible code:
        ....
        return nil; // Should the analyzer cover this piece of code? Should we require the cast (__nonnull)nil?
    }
    ....
}
```

There are these directions:

- We can either take the branch; this way the branch is analyzed
- Should we not warn about any nullability issues in that branch? Probably not, it is ok to break the nullability postconditions when the nullability preconditions are violated.
- We can assume that these pointers are not null and we lose coverage with the analyzer. (This can be implemented either in constraint solver or in the checker itself.)

Other Issues to keep in mind/take care of:

- Messaging:
  - Sending a message to a nullable pointer
    - **Even though the method might return a nonnull pointer, when it was sent to a nullable pointer the return type will be nullable.**
      - The result is nullable unless the receiver is known to be non null.
  - Sending a message to a unspecified or nonnull pointer
    - If the pointer is not assumed to be nil, we should be optimistic and use the nullability implied by the method.
      - This will not happen automatically, since the AST will have null unspecified in this case.

Inlining

A symbol may need to be treated differently inside an inlined body. For example, consider these conversions from nonnull to nullable in presence of inlining:

```
id obj = getNonnull();
takesNullable(obj);
takesNonnull(obj);

void takesNullable(nullable id obj) {
   obj->ivar // we should assume obj is nullable and warn here
}
```

With no special treatment, when the takesNullable is inlined the analyzer will not warn when the obj symbol is dereferenced. One solution for this is to reanalyze takesNullable as a top level function to get possible violations. The alternative method, deducing nullability information from the arguments after inlining is not robust enough (for example there might be more parameters with different nullability, but in the given path the two parameters might end up being the same symbol or there can be nested functions that take different view of the nullability of the same symbol). So the symbol will remain nonnull to avoid false positives but the functions that takes nullable parameters will be analyzed separately as well without inlining.

Annotations on multi level pointers

Tracking multiple levels of annotations for pointers pointing to pointers would make the checker more complicated, because this way a vector of nullability qualifiers would be needed to be tracked for each symbol. This is not a big caveat, since once the top level pointer is dereferenced, the symvol for the inner pointer will have the nullability information. The lack of multi level annotation tracking only observable, when multiple levels of pointers are passed to a function which has a parameter with multiple levels of annotations. So for now the checker support the top level nullability qualifiers only.:

```
int * __nonnull * __nullable p;
int ** q = p;
takesStarNullableStarNullable(q);
```

Implementation notes

What to track?

- The checker would track memory regions, and to each relevant region a qualifier information would be attached which is either nullable, nonnull or null unspecified (or contradicted to suppress warnings for a specific region).
- On a branch, where a nullable pointer is known to be non null, the checker treat it as a same way as a pointer annotated as nonnull.
- When there is an explicit cast from a null unspecified to either nonnull or nullable I will trust the cast.
- Unannotated pointers are treated the same way as pointers annotated with nullability unspecified qualifier, unless the region is wrapped in ASSUME_NONNULL macros.

- We might want to implement a callback for entry points to top level functions, where the pointer nullability assumptions would be made.

Region Store

The analyzer “Store” represents the contents of memory regions. It is an opaque functional data structure stored in each `ProgramState`; the only class that can modify the store is its associated StoreManager.

Currently (Feb. 2013), the only StoreManager implementation being used is `RegionStoreManager`. This store records bindings to memory regions using a “base region + offset” key. (This allows `*p` and `p[0]` to map to the same location, among other benefits.)

Regions are grouped into “clusters”, which roughly correspond to “regions with the same base region”. This allows certain operations to be more efficient, such as invalidation.

Regions that do not have a known offset use a special “symbolic” offset. These keys store both the original region, and the “concrete offset region” – the last region whose offset is entirely concrete. (For example, in the expression `foo.bar[1][i].baz`, the concrete offset region is the array `foo.bar[1]`, since that has a known offset from the start of the top-level `foo` struct.)

Binding Invalidation

Supporting both concrete and symbolic offsets makes things a bit tricky. Here’s an example:

```
foo[0] = 0;
foo[1] = 1;
foo[i] = i;
```

After the third assignment, nothing can be said about the value of `foo[0]`, because `foo[i]` may have overwritten it! Thus, *binding to a region with a symbolic offset invalidates the entire concrete offset region.* We know `foo[i]` is somewhere within `foo`, so we don’t have to invalidate anything else, but we do have to be conservative about all other bindings within `foo`.

Continuing the example:

```
foo[i] = i;
foo[0] = 0;
```

After this latest assignment, nothing can be said about the value of `foo[i]`, because `foo[0]` may have overwritten it! *Binding to a region R with a concrete offset invalidates any symbolic offset bindings whose concrete offset region is a super-region **or** sub-region of R.* All we know about `foo[i]` is that it is somewhere within `foo`, so changing *anything* within `foo` might change `foo[i]`, and changing *all* of `foo` (or its base region) will *definitely* change `foo[i]`.

This logic could be improved by using the current constraints on `i`, at the cost of speed. The latter case could also be improved by matching region kinds, i.e. changing `foo[0].a` is unlikely to affect `foo[i].b`, no matter what `i` is.

For more detail, read through `RegionStoreManager::removeSubRegionBindings` in RegionStore.cpp.

ObjCIvarRegions

Objective-C instance variables require a bit of special handling. Like struct fields, they are not base regions, and when their parent object region is invalidated, all the instance variables must be invalidated as well. However, they have no concrete compile-time offsets (in the modern, “non-fragile” runtime), and so cannot easily be represented as an offset from the start of the object in the analyzer. Moreover, this means that invalidating a single instance variable should *not* invalidate the rest of the object, since unlike struct fields or array elements there is no way to perform pointer arithmetic to access another instance variable.

Consequently, although the base region of an ObjCIvarRegion is the entire object, RegionStore offsets are computed from the start of the instance variable. Thus it is not valid to assume that all bindings with non-symbolic offsets start from the base region!

Region Invalidation

Unlike binding invalidation, region invalidation occurs when the entire contents of a region may have changed—say, because it has been passed to a function the analyzer can model, like memcpy, or because its address has escaped, usually as an argument to an opaque function call. In these cases we need to throw away not just all bindings within the region itself, but within its entire cluster, since neighboring regions may be accessed via pointer arithmetic.

Region invalidation typically does even more than this, however. Because it usually represents the complete escape of a region from the analyzer's model, its *contents* must also be transitively invalidated. (For example, if a region p of type `int **` is invalidated, the contents of `*p` and `**p` may have changed as well.) The algorithm that traverses this transitive closure of accessible regions is known as ClusterAnalysis, and is also used for finding all live bindings in the store (in order to throw away the dead ones). The name "ClusterAnalysis" predates the cluster-based organization of bindings, but refers to the same concept: during invalidation and liveness analysis, all bindings within a cluster must be treated in the same way for a conservative model of program behavior.

Default Bindings

Most bindings in RegionStore are simple scalar values – integers and pointers. These are known as "Direct" bindings. However, RegionStore supports a second type of binding called a "Default" binding. These are used to provide values to all the elements of an aggregate type (struct or array) without having to explicitly specify a binding for each individual element.

When there is no Direct binding for a particular region, the store manager looks at each super-region in turn to see if there is a Default binding. If so, this value is used as the value of the original region. The search ends when the base region is reached, at which point the RegionStore will pick an appropriate default value for the region (usually a symbolic value, but sometimes zero, for static data, or "uninitialized", for stack variables).

```
int manyInts[10];
manyInts[1] = 42;   // Creates a Direct binding for manyInts[1].
print(manyInts[1]); // Retrieves the Direct binding for manyInts[1];
print(manyInts[0]); // There is no Direct binding for manyInts[0].
                    // Is there a Default binding for the entire array?
                    // There is not, but it is a stack variable, so we use
                    // "uninitialized" as the default value (and emit a
                    // diagnostic!).
```

NOTE: The fact that bindings are stored as a base region plus an offset limits the Default Binding strategy, because in C aggregates can contain other aggregates. In the current implementation of RegionStore, there is no way to distinguish a Default binding for an entire aggregate from a Default binding for the sub-aggregate at offset 0.

Lazy Bindings (LazyCompoundVal)

RegionStore implements an optimization for copying aggregates (structs and arrays) called "lazy bindings", implemented using a special SVal called LazyCompoundVal. When the store is asked for the "binding" for an entire aggregate (i.e. for an lvalue-to-rvalue conversion), it returns a LazyCompoundVal instead. When this value is then stored into a variable, it is bound as a Default value. This makes copying arrays and structs much cheaper than if they had required memberwise access.

Under the hood, a LazyCompoundVal is implemented as a uniqued pair of (region, store), representing "the value of the region during this 'snapshot' of the store". This has important implications for any sort of liveness or reachability analysis, which must take the bindings in the old store into account.

Retrieving a value from a lazy binding happens in the same way as any other Default binding: since there is no direct binding, the store manager falls back to super-regions to look for an appropriate default binding. LazyCompoundVal differs from a normal default binding, however, in that it contains several different values, instead of one value that will appear several times. Because of this, the store manager has to reconstruct the subregion chain on top of the LazyCompoundVal region, and look up *that* region in the previous store.

Here's a concrete example:

```
CGPoint p;
p.x = 42;       // A Direct binding is made to the FieldRegion 'p.x'.
CGPoint p2 = p; // A LazyCompoundVal is created for 'p', along with a
                // snapshot of the current store state. This value is then
                // used as a Default binding for the VarRegion 'p2'.
```

```
return p2.x;    // The binding for FieldRegion 'p2.x' is requested.
                // There is no Direct binding, so we look for a Default
                // binding to 'p2' and find the LCV.
                // Because it's a LCV, we look at our requested region
                // and see that it's the '.x' field. We ask for the value
                // of 'p.x' within the snapshot, and get back 42.
```

Thread Safety Analysis

Introduction

Clang Thread Safety Analysis is a C++ language extension which warns about potential race conditions in code. The analysis is completely static (i.e. compile-time); there is no run-time overhead. The analysis is still under active development, but it is mature enough to be deployed in an industrial setting. It is being developed by Google, in collaboration with CERT/SEI, and is used extensively in Google's internal code base.

Thread safety analysis works very much like a type system for multi-threaded programs. In addition to declaring the *type* of data (e.g. `int`, `float`, etc.), the programmer can (optionally) declare how access to that data is controlled in a multi-threaded environment. For example, if `foo` is *guarded by* the mutex `mu`, then the analysis will issue a warning whenever a piece of code reads or writes to `foo` without first locking `mu`. Similarly, if there are particular routines that should only be called by the GUI thread, then the analysis will warn if other threads call those routines.

Getting Started

```
#include "mutex.h"

class BankAccount {
private:
  Mutex mu;
  int   balance GUARDED_BY(mu);

  void depositImpl(int amount) {
    balance += amount;       // WARNING! Cannot write balance without locking mu.
  }

  void withdrawImpl(int amount) REQUIRES(mu) {
    balance -= amount;       // OK. Caller must have locked mu.
  }

public:
  void withdraw(int amount) {
    mu.Lock();
    withdrawImpl(amount);    // OK.  We've locked mu.
  }                          // WARNING!  Failed to unlock mu.

  void transferFrom(BankAccount& b, int amount) {
    mu.Lock();
    b.withdrawImpl(amount);  // WARNING!  Calling withdrawImpl() requires locking b.mu.
    depositImpl(amount);     // OK.  depositImpl() has no requirements.
    mu.Unlock();
  }
};
```

This example demonstrates the basic concepts behind the analysis. The `GUARDED_BY` attribute declares that a thread must lock `mu` before it can read or write to `balance`, thus ensuring that the increment and decrement operations are atomic. Similarly, `REQUIRES` declares that the calling thread must lock `mu` before calling `withdrawImpl`. Because the caller is assumed to have locked `mu`, it is safe to modify `balance` within the body of the method.

The `depositImpl()` method does not have `REQUIRES`, so the analysis issues a warning. Thread safety analysis is not inter-procedural, so caller requirements must be explicitly declared. There is also a warning in `transferFrom()`, because although the method locks `this->mu`, it does not lock `b.mu`. The analysis understands that these are two separate mutexes, in two different objects.

Finally, there is a warning in the `withdraw()` method, because it fails to unlock `mu`. Every lock must have a corresponding unlock, and the analysis will detect both double locks, and double unlocks. A function is allowed to acquire a lock without releasing it, (or vice versa), but it must be annotated as such (using `ACQUIRE`/`RELEASE`).

Running The Analysis

To run the analysis, simply compile with the `-Wthread-safety` flag, e.g.

```
clang -c -Wthread-safety example.cpp
```

Note that this example assumes the presence of a suitably annotated mutex.h that declares which methods perform locking, unlocking, and so on.

Basic Concepts: Capabilities

Thread safety analysis provides a way of protecting *resources* with *capabilities*. A resource is either a data member, or a function/method that provides access to some underlying resource. The analysis ensures that the calling thread cannot access the *resource* (i.e. call the function, or read/write the data) unless it has the *capability* to do so.

Capabilities are associated with named C++ objects which declare specific methods to acquire and release the capability. The name of the object serves to identify the capability. The most common example is a mutex. For example, if `mu` is a mutex, then calling `mu.Lock()` causes the calling thread to acquire the capability to access data that is protected by `mu`. Similarly, calling `mu.Unlock()` releases that capability.

A thread may hold a capability either *exclusively* or *shared*. An exclusive capability can be held by only one thread at a time, while a shared capability can be held by many threads at the same time. This mechanism enforces a multiple-reader, single-writer pattern. Write operations to protected data require exclusive access, while read operations require only shared access.

At any given moment during program execution, a thread holds a specific set of capabilities (e.g. the set of mutexes that it has locked.) These act like keys or tokens that allow the thread to access a given resource. Just like physical security keys, a thread cannot make copy of a capability, nor can it destroy one. A thread can only release a capability to another thread, or acquire one from another thread. The annotations are deliberately agnostic about the exact mechanism used to acquire and release capabilities; it assumes that the underlying implementation (e.g. the Mutex implementation) does the handoff in an appropriate manner.

The set of capabilities that are actually held by a given thread at a given point in program execution is a run-time concept. The static analysis works by calculating an approximation of that set, called the *capability environment*. The capability environment is calculated for every program point, and describes the set of capabilities that are statically known to be held, or not held, at that particular point. This environment is a conservative approximation of the full set of capabilities that will actually held by a thread at run-time.

Reference Guide

The thread safety analysis uses attributes to declare threading constraints. Attributes must be attached to named declarations, such as classes, methods, and data members. Users are *strongly advised* to define macros for the various attributes; example definitions can be found in mutex.h, below. The following documentation assumes the use of macros.

The attributes only control assumptions made by thread safety analysis and the warnings it issues. They don't affect generated code or behavior at run-time.

For historical reasons, prior versions of thread safety used macro names that were very lock-centric. These macros have since been renamed to fit a more general capability model. The prior names are still in use, and will be mentioned under the tag *previously* where appropriate.

GUARDED_BY(c) and PT_GUARDED_BY(c)

`GUARDED_BY` is an attribute on data members, which declares that the data member is protected by the given capability. Read operations on the data require shared access, while write operations require exclusive access.

`PT_GUARDED_BY` is similar, but is intended for use on pointers and smart pointers. There is no constraint on the data member itself, but the *data that it points to* is protected by the given capability.

```
Mutex mu;
int *p1             GUARDED_BY(mu);
int *p2             PT_GUARDED_BY(mu);
unique_ptr<int> p3  PT_GUARDED_BY(mu);

void test() {
  p1 = 0;             // Warning!

  *p2 = 42;           // Warning!
  p2 = new int;       // OK.

  *p3 = 42;           // Warning!
  p3.reset(new int);  // OK.
}
```

REQUIRES(...), REQUIRES_SHARED(...)

Previously: `EXCLUSIVE_LOCKS_REQUIRED`, `SHARED_LOCKS_REQUIRED`

`REQUIRES` is an attribute on functions or methods, which declares that the calling thread must have exclusive access to the given capabilities. More than one capability may be specified. The capabilities must be held on entry to the function, *and must still be held on exit*.

`REQUIRES_SHARED` is similar, but requires only shared access.

```
Mutex mu1, mu2;
int a GUARDED_BY(mu1);
int b GUARDED_BY(mu2);

void foo() REQUIRES(mu1, mu2) {
  a = 0;
  b = 0;
}

void test() {
  mu1.Lock();
  foo();         // Warning!  Requires mu2.
  mu1.Unlock();
}
```

ACQUIRE(...), ACQUIRE_SHARED(...), RELEASE(...), RELEASE_SHARED(...), RELEASE_GENERIC(...)

Previously: `EXCLUSIVE_LOCK_FUNCTION`, `SHARED_LOCK_FUNCTION`, `UNLOCK_FUNCTION`

`ACQUIRE` and `ACQUIRE_SHARED` are attributes on functions or methods declaring that the function acquires a capability, but does not release it. The given capability must not be held on entry, and will be held on exit (exclusively for `ACQUIRE`, shared for `ACQUIRE_SHARED`).

`RELEASE`, `RELEASE_SHARED`, and `RELEASE_GENERIC` declare that the function releases the given capability. The capability must be held on entry (exclusively for `RELEASE`, shared for `RELEASE_SHARED`, exclusively or shared for `RELEASE_GENERIC`), and will no longer be held on exit.

```
Mutex mu;
MyClass myObject GUARDED_BY(mu);
```

```
void lockAndInit() ACQUIRE(mu) {
  mu.Lock();
  myObject.init();
}

void cleanupAndUnlock() RELEASE(mu) {
  myObject.cleanup();
}                          // Warning!  Need to unlock mu.

void test() {
  lockAndInit();
  myObject.doSomething();
  cleanupAndUnlock();
  myObject.doSomething();  // Warning, mu is not locked.
}
```

If no argument is passed to `ACQUIRE` or `RELEASE`, then the argument is assumed to be `this`, and the analysis will not check the body of the function. This pattern is intended for use by classes which hide locking details behind an abstract interface. For example:

```
template <class T>
class CAPABILITY("mutex") Container {
private:
  Mutex mu;
  T* data;

public:
  // Hide mu from public interface.
  void Lock()   ACQUIRE() { mu.Lock(); }
  void Unlock() RELEASE() { mu.Unlock(); }

  T& getElem(int i) { return data[i]; }
};

void test() {
  Container<int> c;
  c.Lock();
  int i = c.getElem(0);
  c.Unlock();
}
```

EXCLUDES(...)

Previously: `LOCKS_EXCLUDED`

`EXCLUDES` is an attribute on functions or methods, which declares that the caller must *not* hold the given capabilities. This annotation is used to prevent deadlock. Many mutex implementations are not re-entrant, so deadlock can occur if the function acquires the mutex a second time.

```
Mutex mu;
int a GUARDED_BY(mu);

void clear() EXCLUDES(mu) {
  mu.Lock();
  a = 0;
  mu.Unlock();
}

void reset() {
  mu.Lock();
  clear();     // Warning!  Caller cannot hold 'mu'.
```

```
  mu.Unlock();
}
```

Unlike REQUIRES, EXCLUDES is optional. The analysis will not issue a warning if the attribute is missing, which can lead to false negatives in some cases. This issue is discussed further in Negative Capabilities.

NO_THREAD_SAFETY_ANALYSIS

NO_THREAD_SAFETY_ANALYSIS is an attribute on functions or methods, which turns off thread safety checking for that method. It provides an escape hatch for functions which are either (1) deliberately thread-unsafe, or (2) are thread-safe, but too complicated for the analysis to understand. Reasons for (2) will be described in the Known Limitations, below.

```
class Counter {
  Mutex mu;
  int a GUARDED_BY(mu);

  void unsafeIncrement() NO_THREAD_SAFETY_ANALYSIS { a++; }
};
```

Unlike the other attributes, NO_THREAD_SAFETY_ANALYSIS is not part of the interface of a function, and should thus be placed on the function definition (in the .cc or .cpp file) rather than on the function declaration (in the header).

RETURN_CAPABILITY(c)

Previously: LOCK_RETURNED

RETURN_CAPABILITY is an attribute on functions or methods, which declares that the function returns a reference to the given capability. It is used to annotate getter methods that return mutexes.

```
class MyClass {
private:
  Mutex mu;
  int a GUARDED_BY(mu);

public:
  Mutex* getMu() RETURN_CAPABILITY(mu) { return &mu; }

  // analysis knows that getMu() == mu
  void clear() REQUIRES(getMu()) { a = 0; }
};
```

ACQUIRED_BEFORE(...), ACQUIRED_AFTER(...)

ACQUIRED_BEFORE and ACQUIRED_AFTER are attributes on member declarations, specifically declarations of mutexes or other capabilities. These declarations enforce a particular order in which the mutexes must be acquired, in order to prevent deadlock.

```
Mutex m1;
Mutex m2 ACQUIRED_AFTER(m1);

// Alternative declaration
// Mutex m2;
// Mutex m1 ACQUIRED_BEFORE(m2);

void foo() {
  m2.Lock();
  m1.Lock();  // Warning!  m2 must be acquired after m1.
  m1.Unlock();
  m2.Unlock();
}
```

CAPABILITY(<string>)

Previously: `LOCKABLE`

`CAPABILITY` is an attribute on classes, which specifies that objects of the class can be used as a capability. The string argument specifies the kind of capability in error messages, e.g. `"mutex"`. See the `Container` example given above, or the `Mutex` class in mutex.h.

SCOPED_CAPABILITY

Previously: `SCOPED_LOCKABLE`

`SCOPED_CAPABILITY` is an attribute on classes that implement RAII-style locking, in which a capability is acquired in the constructor, and released in the destructor. Such classes require special handling because the constructor and destructor refer to the capability via different names; see the `MutexLocker` class in mutex.h, below.

Scoped capabilities are treated as capabilities that are implicitly acquired on construction and released on destruction. They are associated with the set of (regular) capabilities named in thread safety attributes on the constructor. Acquire-type attributes on other member functions are treated as applying to that set of associated capabilities, while `RELEASE` implies that a function releases all associated capabilities in whatever mode they're held.

TRY_ACQUIRE(<bool>, ...), TRY_ACQUIRE_SHARED(<bool>, ...)

Previously: `EXCLUSIVE_TRYLOCK_FUNCTION`, `SHARED_TRYLOCK_FUNCTION`

These are attributes on a function or method that tries to acquire the given capability, and returns a boolean value indicating success or failure. The first argument must be `true` or `false`, to specify which return value indicates success, and the remaining arguments are interpreted in the same way as `ACQUIRE`. See mutex.h, below, for example uses.

Because the analysis doesn't support conditional locking, a capability is treated as acquired after the first branch on the return value of a try-acquire function.

```
Mutex mu;
int a GUARDED_BY(mu);

void foo() {
  bool success = mu.TryLock();
  a = 0;          // Warning, mu is not locked.
  if (success) {
    a = 0;        // Ok.
    mu.Unlock();
  } else {
    a = 0;        // Warning, mu is not locked.
  }
}
```

ASSERT_CAPABILITY(...) and ASSERT_SHARED_CAPABILITY(...)

Previously: `ASSERT_EXCLUSIVE_LOCK`, `ASSERT_SHARED_LOCK`

These are attributes on a function or method which asserts the calling thread already holds the given capability, for example by performing a run-time test and terminating if the capability is not held. Presence of this annotation causes the analysis to assume the capability is held after calls to the annotated function. See mutex.h, below, for example uses.

GUARDED_VAR and PT_GUARDED_VAR

Use of these attributes has been deprecated.

Warning flags

- `-Wthread-safety`: Umbrella flag which turns on the following:

- `-Wthread-safety-attributes`: Semantic checks for thread safety attributes.
- `-Wthread-safety-analysis`: The core analysis.
- `-Wthread-safety-precise`**: Requires that mutex expressions match precisely.**
  This warning can be disabled for code which has a lot of aliases.
- `-Wthread-safety-reference`: Checks when guarded members are passed by reference.

Negative Capabilities are an experimental feature, which are enabled with:

- `-Wthread-safety-negative`: Negative capabilities. Off by default.

When new features and checks are added to the analysis, they can often introduce additional warnings. Those warnings are initially released as *beta* warnings for a period of time, after which they are migrated into the standard analysis.

- `-Wthread-safety-beta`: New features. Off by default.

Negative Capabilities

Thread Safety Analysis is designed to prevent both race conditions and deadlock. The GUARDED_BY and REQUIRES attributes prevent race conditions, by ensuring that a capability is held before reading or writing to guarded data, and the EXCLUDES attribute prevents deadlock, by making sure that a mutex is *not* held.

However, EXCLUDES is an optional attribute, and does not provide the same safety guarantee as REQUIRES. In particular:

- A function which acquires a capability does not have to exclude it.
- A function which calls a function that excludes a capability does not have transitively exclude that capability.

As a result, EXCLUDES can easily produce false negatives:

```
class Foo {
  Mutex mu;

  void foo() {
    mu.Lock();
    bar();           // No warning.
    baz();           // No warning.
    mu.Unlock();
  }

  void bar() {       // No warning.  (Should have EXCLUDES(mu)).
    mu.Lock();
    // ...
    mu.Unlock();
  }

  void baz() {
    bif();           // No warning.  (Should have EXCLUDES(mu)).
  }

  void bif() EXCLUDES(mu);
};
```

Negative requirements are an alternative EXCLUDES that provide a stronger safety guarantee. A negative requirement uses the REQUIRES attribute, in conjunction with the `!` operator, to indicate that a capability should *not* be held.

For example, using `REQUIRES(!mu)` instead of `EXCLUDES(mu)` will produce the appropriate warnings:

```
class FooNeg {
  Mutex mu;
```

```
  void foo() REQUIRES(!mu) {   // foo() now requires !mu.
    mu.Lock();
    bar();
    baz();
    mu.Unlock();
  }

  void bar() {
    mu.Lock();       // WARNING!  Missing REQUIRES(!mu).
    // ...
    mu.Unlock();
  }

  void baz() {
    bif();           // WARNING!  Missing REQUIRES(!mu).
  }

  void bif() REQUIRES(!mu);
};
```

Negative requirements are an experimental feature which is off by default, because it will produce many warnings in existing code. It can be enabled by passing `-Wthread-safety-negative`.

Frequently Asked Questions

`Q.` Should I put attributes in the header file, or in the .cc/.cpp/.cxx file?

(A) Attributes are part of the formal interface of a function, and should always go in the header, where they are visible to anything that includes the header. Attributes in the .cpp file are not visible outside of the immediate translation unit, which leads to false negatives and false positives.

`Q.` "*Mutex is not locked on every path through here?*" What does that mean?

`A.` See No conditionally held locks., below.

Known Limitations

Lexical scope

Thread safety attributes contain ordinary C++ expressions, and thus follow ordinary C++ scoping rules. In particular, this means that mutexes and other capabilities must be declared before they can be used in an attribute. Use-before-declaration is okay within a single class, because attributes are parsed at the same time as method bodies. (C++ delays parsing of method bodies until the end of the class.) However, use-before-declaration is not allowed between classes, as illustrated below.

```
class Foo;

class Bar {
  void bar(Foo* f) REQUIRES(f->mu);  // Error: mu undeclared.
};

class Foo {
  Mutex mu;
};
```

Private Mutexes

Good software engineering practice dictates that mutexes should be private members, because the locking mechanism used by a thread-safe class is part of its internal implementation. However, private mutexes can sometimes leak into the public interface of a class. Thread safety attributes follow normal C++ access restrictions, so if `mu` is a private member of `c`, then it is an error to write `c.mu` in an attribute.

One workaround is to (ab)use the `RETURN_CAPABILITY` attribute to provide a public *name* for a private mutex, without actually exposing the underlying mutex. For example:

```
class MyClass {
private:
  Mutex mu;

public:
  // For thread safety analysis only.  Does not need to be defined.
  Mutex* getMu() RETURN_CAPABILITY(mu);

  void doSomething() REQUIRES(mu);
};

void doSomethingTwice(MyClass& c) REQUIRES(c.getMu()) {
  // The analysis thinks that c.getMu() == c.mu
  c.doSomething();
  c.doSomething();
}
```

In the above example, `doSomethingTwice()` is an external routine that requires `c.mu` to be locked, which cannot be declared directly because `mu` is private. This pattern is discouraged because it violates encapsulation, but it is sometimes necessary, especially when adding annotations to an existing code base. The workaround is to define `getMu()` as a fake getter method, which is provided only for the benefit of thread safety analysis.

No conditionally held locks.

The analysis must be able to determine whether a lock is held, or not held, at every program point. Thus, sections of code where a lock *might be held* will generate spurious warnings (false positives). For example:

```
void foo() {
  bool b = needsToLock();
  if (b) mu.Lock();
  ...  // Warning!  Mutex 'mu' is not held on every path through here.
  if (b) mu.Unlock();
}
```

No checking inside constructors and destructors.

The analysis currently does not do any checking inside constructors or destructors. In other words, every constructor and destructor is treated as if it was annotated with `NO_THREAD_SAFETY_ANALYSIS`. The reason for this is that during initialization, only one thread typically has access to the object which is being initialized, and it is thus safe (and common practice) to initialize guarded members without acquiring any locks. The same is true of destructors.

Ideally, the analysis would allow initialization of guarded members inside the object being initialized or destroyed, while still enforcing the usual access restrictions on everything else. However, this is difficult to enforce in practice, because in complex pointer-based data structures, it is hard to determine what data is owned by the enclosing object.

No inlining.

Thread safety analysis is strictly intra-procedural, just like ordinary type checking. It relies only on the declared attributes of a function, and will not attempt to inline any method calls. As a result, code such as the following will not work:

```
template<class T>
class AutoCleanup {
  T* object;
  void (T::*mp)();

public:
  AutoCleanup(T* obj, void (T::*imp)()) : object(obj), mp(imp) { }
  ~AutoCleanup() { (object->*mp)(); }
```

```
};

Mutex mu;
void foo() {
  mu.Lock();
  AutoCleanup<Mutex>(&mu, &Mutex::Unlock);
  // ...
}  // Warning, mu is not unlocked.
```

In this case, the destructor of `Autocleanup` calls `mu.Unlock()`, so the warning is bogus. However, thread safety analysis cannot see the unlock, because it does not attempt to inline the destructor. Moreover, there is no way to annotate the destructor, because the destructor is calling a function that is not statically known. This pattern is simply not supported.

No alias analysis.

The analysis currently does not track pointer aliases. Thus, there can be false positives if two pointers both point to the same mutex.

```
class MutexUnlocker {
  Mutex* mu;

public:
  MutexUnlocker(Mutex* m) RELEASE(m) : mu(m)  { mu->Unlock(); }
  ~MutexUnlocker() ACQUIRE(mu) { mu->Lock(); }
};

Mutex mutex;
void test() REQUIRES(mutex) {
  {
    MutexUnlocker munl(&mutex);  // unlocks mutex
    doSomeIO();
  }                              // Warning: locks munl.mu
}
```

The MutexUnlocker class is intended to be the dual of the MutexLocker class, defined in mutex.h. However, it doesn't work because the analysis doesn't know that munl.mu == mutex. The SCOPED_CAPABILITY attribute handles aliasing for MutexLocker, but does so only for that particular pattern.

ACQUIRED_BEFORE(...) and ACQUIRED_AFTER(...) are currently unimplemented.

To be fixed in a future update.

mutex.h

Thread safety analysis can be used with any threading library, but it does require that the threading API be wrapped in classes and methods which have the appropriate annotations. The following code provides `mutex.h` as an example; these methods should be filled in to call the appropriate underlying implementation.

```
#ifndef THREAD_SAFETY_ANALYSIS_MUTEX_H
#define THREAD_SAFETY_ANALYSIS_MUTEX_H

// Enable thread safety attributes only with clang.
// The attributes can be safely erased when compiling with other compilers.
#if defined(__clang__) && (!defined(SWIG))
#define THREAD_ANNOTATION_ATTRIBUTE__(x)   __attribute__((x))
#else
#define THREAD_ANNOTATION_ATTRIBUTE__(x)   // no-op
#endif

#define CAPABILITY(x) \
```

```
  THREAD_ANNOTATION_ATTRIBUTE__(capability(x))

#define SCOPED_CAPABILITY \
  THREAD_ANNOTATION_ATTRIBUTE__(scoped_lockable)

#define GUARDED_BY(x) \
  THREAD_ANNOTATION_ATTRIBUTE__(guarded_by(x))

#define PT_GUARDED_BY(x) \
  THREAD_ANNOTATION_ATTRIBUTE__(pt_guarded_by(x))

#define ACQUIRED_BEFORE(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(acquired_before(__VA_ARGS__))

#define ACQUIRED_AFTER(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(acquired_after(__VA_ARGS__))

#define REQUIRES(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(requires_capability(__VA_ARGS__))

#define REQUIRES_SHARED(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(requires_shared_capability(__VA_ARGS__))

#define ACQUIRE(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(acquire_capability(__VA_ARGS__))

#define ACQUIRE_SHARED(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(acquire_shared_capability(__VA_ARGS__))

#define RELEASE(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(release_capability(__VA_ARGS__))

#define RELEASE_SHARED(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(release_shared_capability(__VA_ARGS__))

#define RELEASE_GENERIC(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(release_generic_capability(__VA_ARGS__))

#define TRY_ACQUIRE(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(try_acquire_capability(__VA_ARGS__))

#define TRY_ACQUIRE_SHARED(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(try_acquire_shared_capability(__VA_ARGS__))

#define EXCLUDES(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(locks_excluded(__VA_ARGS__))

#define ASSERT_CAPABILITY(x) \
  THREAD_ANNOTATION_ATTRIBUTE__(assert_capability(x))

#define ASSERT_SHARED_CAPABILITY(x) \
  THREAD_ANNOTATION_ATTRIBUTE__(assert_shared_capability(x))

#define RETURN_CAPABILITY(x) \
  THREAD_ANNOTATION_ATTRIBUTE__(lock_returned(x))

#define NO_THREAD_SAFETY_ANALYSIS \
  THREAD_ANNOTATION_ATTRIBUTE__(no_thread_safety_analysis)
```

```
// Defines an annotated interface for mutexes.
// These methods can be implemented to use any internal mutex implementation.
class CAPABILITY("mutex") Mutex {
public:
  // Acquire/lock this mutex exclusively.  Only one thread can have exclusive
  // access at any one time.  Write operations to guarded data require an
  // exclusive lock.
  void Lock() ACQUIRE();

  // Acquire/lock this mutex for read operations, which require only a shared
  // lock.  This assumes a multiple-reader, single writer semantics.  Multiple
  // threads may acquire the mutex simultaneously as readers, but a writer
  // must wait for all of them to release the mutex before it can acquire it
  // exclusively.
  void ReaderLock() ACQUIRE_SHARED();

  // Release/unlock an exclusive mutex.
  void Unlock() RELEASE();

  // Release/unlock a shared mutex.
  void ReaderUnlock() RELEASE_SHARED();

  // Generic unlock, can unlock exclusive and shared mutexes.
  void GenericUnlock() RELEASE_GENERIC();

  // Try to acquire the mutex.  Returns true on success, and false on failure.
  bool TryLock() TRY_ACQUIRE(true);

  // Try to acquire the mutex for read operations.
  bool ReaderTryLock() TRY_ACQUIRE_SHARED(true);

  // Assert that this mutex is currently held by the calling thread.
  void AssertHeld() ASSERT_CAPABILITY(this);

  // Assert that is mutex is currently held for read operations.
  void AssertReaderHeld() ASSERT_SHARED_CAPABILITY(this);

  // For negative capabilities.
  const Mutex& operator!() const { return *this; }
};

// Tag types for selecting a constructor.
struct adopt_lock_t {} inline constexpr adopt_lock = {};
struct defer_lock_t {} inline constexpr defer_lock = {};
struct shared_lock_t {} inline constexpr shared_lock = {};

// MutexLocker is an RAII class that acquires a mutex in its constructor, and
// releases it in its destructor.
class SCOPED_CAPABILITY MutexLocker {
private:
  Mutex* mut;
  bool locked;

public:
  // Acquire mu, implicitly acquire *this and associate it with mu.
  MutexLocker(Mutex *mu) ACQUIRE(mu) : mut(mu), locked(true) {
    mu->Lock();
  }

  // Assume mu is held, implicitly acquire *this and associate it with mu.
```

```
  MutexLocker(Mutex *mu, adopt_lock_t) REQUIRES(mu) : mut(mu), locked(true) {}

  // Acquire mu in shared mode, implicitly acquire *this and associate it with mu.
  MutexLocker(Mutex *mu, shared_lock_t) ACQUIRE_SHARED(mu) : mut(mu), locked(true) {
    mu->ReaderLock();
  }

  // Assume mu is held in shared mode, implicitly acquire *this and associate it with mu.
  MutexLocker(Mutex *mu, adopt_lock_t, shared_lock_t) REQUIRES_SHARED(mu)
    : mut(mu), locked(true) {}

  // Assume mu is not held, implicitly acquire *this and associate it with mu.
  MutexLocker(Mutex *mu, defer_lock_t) EXCLUDES(mu) : mut(mu), locked(false) {}

  // Release *this and all associated mutexes, if they are still held.
  // There is no warning if the scope was already unlocked before.
  ~MutexLocker() RELEASE() {
    if (locked)
      mut->GenericUnlock();
  }

  // Acquire all associated mutexes exclusively.
  void Lock() ACQUIRE() {
    mut->Lock();
    locked = true;
  }

  // Try to acquire all associated mutexes exclusively.
  bool TryLock() TRY_ACQUIRE(true) {
    return locked = mut->TryLock();
  }

  // Acquire all associated mutexes in shared mode.
  void ReaderLock() ACQUIRE_SHARED() {
    mut->ReaderLock();
    locked = true;
  }

  // Try to acquire all associated mutexes in shared mode.
  bool ReaderTryLock() TRY_ACQUIRE_SHARED(true) {
    return locked = mut->ReaderTryLock();
  }

  // Release all associated mutexes. Warn on double unlock.
  void Unlock() RELEASE() {
    mut->Unlock();
    locked = false;
  }

  // Release all associated mutexes. Warn on double unlock.
  void ReaderUnlock() RELEASE() {
    mut->ReaderUnlock();
    locked = false;
  }
};


#ifdef USE_LOCK_STYLE_THREAD_SAFETY_ATTRIBUTES
// The original version of thread safety analysis the following attribute
// definitions.  These use a lock-based terminology.  They are still in use
```

```
// by existing thread safety code, and will continue to be supported.

// Deprecated.
#define PT_GUARDED_VAR \
  THREAD_ANNOTATION_ATTRIBUTE__(pt_guarded_var)

// Deprecated.
#define GUARDED_VAR \
  THREAD_ANNOTATION_ATTRIBUTE__(guarded_var)

// Replaced by REQUIRES
#define EXCLUSIVE_LOCKS_REQUIRED(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(exclusive_locks_required(__VA_ARGS__))

// Replaced by REQUIRES_SHARED
#define SHARED_LOCKS_REQUIRED(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(shared_locks_required(__VA_ARGS__))

// Replaced by CAPABILITY
#define LOCKABLE \
  THREAD_ANNOTATION_ATTRIBUTE__(lockable)

// Replaced by SCOPED_CAPABILITY
#define SCOPED_LOCKABLE \
  THREAD_ANNOTATION_ATTRIBUTE__(scoped_lockable)

// Replaced by ACQUIRE
#define EXCLUSIVE_LOCK_FUNCTION(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(exclusive_lock_function(__VA_ARGS__))

// Replaced by ACQUIRE_SHARED
#define SHARED_LOCK_FUNCTION(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(shared_lock_function(__VA_ARGS__))

// Replaced by RELEASE and RELEASE_SHARED
#define UNLOCK_FUNCTION(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(unlock_function(__VA_ARGS__))

// Replaced by TRY_ACQUIRE
#define EXCLUSIVE_TRYLOCK_FUNCTION(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(exclusive_trylock_function(__VA_ARGS__))

// Replaced by TRY_ACQUIRE_SHARED
#define SHARED_TRYLOCK_FUNCTION(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(shared_trylock_function(__VA_ARGS__))

// Replaced by ASSERT_CAPABILITY
#define ASSERT_EXCLUSIVE_LOCK(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(assert_exclusive_lock(__VA_ARGS__))

// Replaced by ASSERT_SHARED_CAPABILITY
#define ASSERT_SHARED_LOCK(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(assert_shared_lock(__VA_ARGS__))

// Replaced by EXCLUDE_CAPABILITY.
#define LOCKS_EXCLUDED(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(locks_excluded(__VA_ARGS__))

// Replaced by RETURN_CAPABILITY
#define LOCK_RETURNED(x) \
```

```
  THREAD_ANNOTATION_ATTRIBUTE__(lock_returned(x))

#endif  // USE_LOCK_STYLE_THREAD_SAFETY_ATTRIBUTES

#endif  // THREAD_SAFETY_ANALYSIS_MUTEX_H
```

Data flow analysis: an informal introduction

Abstract

This document introduces data flow analysis in an informal way. The goal is to give the reader an intuitive understanding of how it works, and show how it applies to a range of refactoring and bug finding problems.

Data flow analysis is a well-established technique; it is described in many papers, books, and videos. If you would like a more formal, or a more thorough explanation of the concepts mentioned in this document, please refer to the following resources:

- The Lattice article in Wikipedia.
- Videos on the PacketPrep YouTube channel that introduce lattices and the necessary background information: #20, #21, #22, #23, #24, #25.
- Introduction to Dataflow Analysis
- Introduction to abstract interpretation.
- Introduction to symbolic execution.
- Static Program Analysis by Anders Møller and Michael I. Schwartzbach.
- EXE: automatically generating inputs of death (a paper that successfully applies symbolic execution to real-world software).

Data flow analysis

The purpose of data flow analysis

Data flow analysis is a static analysis technique that proves facts about a program or its fragment. It can make conclusions about all paths through the program, while taking control flow into account and scaling to large programs. The basic idea is propagating facts about the program through the edges of the control flow graph (CFG) until a fixpoint is reached.

Sample problem and an ad-hoc solution

We would like to explain data flow analysis while discussing an example. Let's imagine that we want to track possible values of an integer variable in our program. Here is how a human could annotate the code:

```
void Example(int n) {
  int x = 0;
  // x is {0}
  if (n > 0) {
    x = 5;
    // x is {5}
  } else {
    x = 42;
    // x is {42}
  }
  // x is {5; 42}
  print(x);
}
```

We use sets of integers to represent possible values of `x`. Local variables have unambiguous values between statements, so we annotate program points between statements with sets of possible values.

Here is how we arrived at these annotations. Assigning a constant to x allows us to make a conclusion that x can only have one value. When control flow from the “then” and “else” branches joins, x can have either value.

Abstract algebra provides a nice formalism that models this kind of structure, namely, a lattice. A join-semilattice is a partially ordered set, in which every two elements have a least upper bound (called a *join*).

`join(a, b) ■ a` and `join(a, b) ■ b` and `join(x, x) = x`

For this problem we will use the lattice of subsets of integers, with set inclusion relation as ordering and set union as a join.

Lattices are often represented visually as Hasse diagrams. Here is a Hasse diagram for our lattice that tracks subsets of integers:

Computing the join in the lattice corresponds to finding the lowest common ancestor (LCA) between two nodes in its Hasse diagram. There is a vast amount of literature on efficiently implementing LCA queries for a DAG, however Efficient Implementation of Lattice Operations (1989) (CiteSeerX, doi) describes a scheme that particularly well-suited for programmatic implementation.

Too much information and “top” values

Let’s try to find the possible sets of values of x in a function that modifies x in a loop:

```
void ExampleOfInfiniteSets() {
  int x = 0; // x is {0}
  while (condition()) {
    x += 1;  // x is {0; 1; 2; …}
  }
  print(x);  // x is {0; 1; 2; …}
}
```

We have an issue: x can have any value greater than zero; that’s an infinite set of values, if the program operated on mathematical integers. In C++ `int` is limited by `INT_MAX` so technically we have a set `{0; 1; …; INT_MAX}` which is still really big.

To make our analysis practical to compute, we have to limit the amount of information that we track. In this case, we can, for example, arbitrarily limit the size of sets to 3 elements. If at a certain program point x has more than 3 possible values, we stop tracking specific values at that program point. Instead, we denote possible values of x with the symbol ■ (pronounced “top” according to a convention in abstract algebra).

```
void ExampleOfTopWithALoop() {
  int x = 0;  // x is {0}
  while (condition()) {
    x += 1;   // x is ■
  }
  print(x);   // x is ■
}
```

The statement “at this program point, x’s possible values are ■” is understood as “at this program point x can have any value because we have too much information, or the information is conflicting”.

Note that we can get more than 3 possible values even without a loop:

```
void ExampleOfTopWithoutLoops(int n) {
  int x = 0;  // x is {0}
  switch(n) {
    case 0:  x = 1; break; // x is {1}
    case 1:  x = 9; break; // x is {9}
    case 2:  x = 7; break; // x is {7}
    default: x = 3; break; // x is {3}
  }
  // x is ■
}
```

Uninitialized variables and “bottom” values

When x is declared but not initialized, it has no possible values. We represent this fact symbolically as ⊥ (pronounced “bottom”).

```
void ExampleOfBottom() {
  int x;    // x is ⊥
  x = 42;   // x is {42}
  print(x);
}
```

Note that using values read from uninitialized variables is undefined behaviour in C++. Generally, compilers and static analysis tools can assume undefined behavior does not happen. We must model uninitialized variables only when we are implementing a checker that specifically is trying to find uninitialized reads. In this example we show how to model uninitialized variables only to demonstrate the concept of “bottom”, and how it applies to possible value analysis. We describe an analysis that finds uninitialized reads in a section below.

A practical lattice that tracks sets of concrete values

Taking into account all corner cases covered above, we can put together a lattice that we can use in practice to track possible values of integer variables. This lattice represents sets of integers with 1, 2, or 3 elements, as well as top and bottom. Here is a Hasse diagram for it:

Formalization

Let’s consider a slightly more complex example, and think about how we can compute the sets of possible values algorithmically.

```
void Example(int n) {
  int x;          // x is ⊥
  if (n > 0) {
    if (n == 42) {
       x = 44;    // x is {44}
    } else {
       x = 5;     // x is {5}
    }
    print(x);     // x is {44; 5}
  } else {
    x = n;        // x is ■
  }
  print(x);       // x is ■
}
```

As humans, we understand the control flow from the program text. We used our understanding of control flow to find program points where two flows join. Formally, control flow is represented by a CFG (control flow graph):

We can compute sets of possible values by propagating them through the CFG of the function:

- When x is declared but not initialized, its possible values are {}. The empty set plays the role of ⊥ in this lattice.
- When x is assigned a concrete value, its possible set of values contains just that specific value.
- When x is assigned some unknown value, it can have any value. We represent this fact as ■.
- When two control flow paths join, we compute the set union of incoming values (limiting the number of elements to 3, representig larger sets as ■).

The sets of possible values are influenced by:

- Statements, for example, assignments.
- Joins in control flow, for example, ones that appear at the end of “if” statements.

Effects of statements are modeled by what is formally known as a transfer function. A transfer function takes two arguments: the statement, and the state of x at the previous program point. It produces the state of x at the next program point. For example, the transfer function for assignment ignores the state at the previous program point:

```
// GIVEN: x is {42; 44}
x = 0;
// CONCLUSION: x is {0}
```

The transfer function for + performs arithmetic on every set member:

```
// GIVEN: x is {42, 44}
x = x + 100;
// CONCLUSION: x is {142, 144}
```

Effects of control flow are modeled by joining the knowledge from all possible previous program points.

```
if (...) {
  ...
  // GIVEN: x is {42}
} else {
  ...
  // GIVEN: x is {44}
}
// CONCLUSION: x is {42; 44}

// GIVEN: x is {42}
while (...) {
  ...
  // GIVEN: x is {44}
}
// CONCLUSION: {42; 44}
```

The predicate that we marked “given” is usually called a precondition, and the conclusion is called a postcondition.

In terms of the CFG, we join the information from all predecessor basic blocks.

Putting it all together, to model the effects of a basic block we compute:

```
out = transfer(basic_block, join(in_1, in_2, ..., in_n))
```

(Note that there are other ways to write this equation that produce higher precision analysis results. The trick is to keep exploring the execution paths separately and delay joining until later. However, we won’t discuss those variations here.)

To make a conclusion about all paths through the program, we repeat this computation on all basic blocks until we reach a fixpoint. In other words, we keep propagating information through the CFG until the computed sets of values stop changing.

If the lattice has a finite height and transfer functions are monotonic the algorithm is guaranteed to terminate. Each iteration of the algorithm can change computed values only to larger values from the lattice. In the worst case, all computed values become ■, which is not very useful, but at least the analysis terminates at that point, because it can’t change any of the values.

Fixpoint iteration can be optimised by only reprocessing basic blocks which had one of their inputs changed on the previous iteration. This is typically implemented using a worklist queue. With this optimisation the time complexity becomes `O(m * |L|)`, where `m` is the number of basic blocks in the CFG and `|L|` is the size of lattice used by the analysis.

Symbolic execution: a very short informal introduction

Symbolic values

In the previous example where we tried to figure out what values a variable can have, the analysis had to be seeded with a concrete value. What if there are no assignments of concrete values in the program? We can still deduce some interesting information by representing unknown input values symbolically, and computing results as symbolic expressions:

```
void PrintAbs(int x) {
  int result;
```

```
  if (x >= 0) {
    result = x;   // result is {x}
  } else {
    result = -x;  // result is {-x}
  }
  print(result);  // result is {x; -x}
}
```

We can't say what specific value gets printed, but we know that it is either `x` or `-x`.

Dataflow analysis is an istance of abstract interpretation, and does not dictate how exactly the lattice and transfer functions should be designed, beyond the necessary conditions for the analysis to converge. Nevertheless, we can use symbolic execution ideas to guide our design of the lattice and transfer functions: lattice values can be symbolic expressions, and transfer functions can construct more complex symbolic expressions from symbolic expressions that represent arguments. See this StackOverflow discussion for a further comparison of abstract interpretation and symbolic execution.

Flow condition

A human can say about the previous example that the function returns `x` when `x >= 0`, and `-x` when `x < 0`. We can make this conclusion programmatically by tracking a flow condition. A flow condition is a predicate written in terms of the program state that is true at a specific program point regardless of the execution path that led to this statement. For example, the flow condition for the program point right before evaluating `result = x` is `x >= 0`.

If we enhance the lattice to be a set of pairs of values and predicates, the dataflow analysis computes the following values:

```
void PrintAbs(int x) {
  int result;
  if (x >= 0) {
    // Flow condition: x >= 0.
    result = x;   // result is {x if x >= 0}
  } else {
    // Flow condition: x < 0.
    result = -x;  // result is {-x if x < 0}
  }
  print(result);  // result is {x if x >= 0; -x if x < 0}
}
```

Of course, in a program with loops, symbolic expressions for flow conditions can grow unbounded. A practical static analysis system must control this growth to keep the symbolic representations manageable and ensure that the data flow analysis terminates. For example, it can use a constraint solver to prune impossible flow conditions, and/or it can abstract them, losing precision, after their symbolic representations grow beyond some threshold. This is similar to how we had to limit the sizes of computed sets of possible values to 3 elements.

Symbolic pointers

This approach proves to be particularly useful for modeling pointer values, since we don't care about specific addresses but just want to give a unique identifier to a memory location.

```
void ExampleOfSymbolicPointers(bool b) {
  int x = 0;     // x is {0}
  int* ptr = &x; // x is {0}      ptr is {&x}
  if (b) {
    *ptr = 42;   // x is {42}     ptr is {&x}
  }
  print(x);      // x is {0; 42}  ptr is {&x}
}
```

Example: finding output parameters

Let's explore how data flow analysis can help with a problem that is hard to solve with other tools in Clang.

Problem description

Output parameters are function parameters of pointer or reference type whose pointee is completely overwritten by the function, and not read before it is overwritten. They are common in pre-C++11 code due to the absence of move semantics. In modern C++ output parameters are non-idiomatic, and return values are used instead.

Imagine that we would like to refactor output parameters to return values to modernize old code. The first step is to identify refactoring candidates through static analysis.

For example, in the following code snippet the pointer c is an output parameter:

```
struct Customer {
  int account_id;
  std::string name;
}

void GetCustomer(Customer *c) {
  c->account_id = ...;
  if (...) {
    c->name = ...;
  } else {
    c->name = ...;
  }
}
```

We would like to refactor this code into:

```
Customer GetCustomer() {
  Customer c;
  c.account_id = ...;
  if (...) {
    c.name = ...;
  } else {
    c.name = ...;
  }
  return c;
}
```

However, in the function below the parameter c is not an output parameter because its field `name` is not overwritten on every path through the function.

```
void GetCustomer(Customer *c) {
  c->account_id = ...;
  if (...) {
    c->name = ...;
  }
}
```

The code also cannot read the value of the parameter before overwriting it:

```
void GetCustomer(Customer *c) {
  use(c->account_id);
  c->name = ...;
  c->account_id = ...;
}
```

Functions that escape the pointer also block the refactoring:

```
Customer* kGlobalCustomer;

void GetCustomer(Customer *c) {
  c->name = ...;
  c->account_id = ...;
  kGlobalCustomer = c;
}
```

To identify a candidate function for refactoring, we need to do the following:

- Find a function with a non-const pointer or reference parameter.
- Find the definition of that function.
- Prove that the function completely overwrites the pointee on all paths before returning.
- Prove that the function reads the pointee only after overwriting it.
- Prove that the function does not persist the pointer in a data structure that is live after the function returns.

There are also requirements that all usage sites of the candidate function must satisfy, for example, that function arguments do not alias, that users are not taking the address of the function, and so on. Let's consider verifying usage site conditions to be a separate static analysis problem.

Lattice design

To analyze the function body we can use a lattice which consists of normal states and failure states. A normal state describes program points where we are sure that no behaviors that block the refactoring have occurred. Normal states keep track of all parameter's member fields that are known to be overwritten on every path from function entry to the corresponding program point. Failure states accumulate observed violations (unsafe reads and pointer escapes) that block the refactoring.

In the partial order of the lattice failure states compare greater than normal states, which guarantees that they "win" when joined with normal states. Order between failure states is determined by inclusion relation on the set of accumulated violations (lattice's ■ is $\subseteq$ on the set of violations). Order between normal states is determined by reversed inclusion relation on the set of overwritten parameter's member fields (lattice's ■ is $\supseteq$ on the set of overwritten fields).

To determine whether a statement reads or writes a field we can implement symbolic evaluation of `DeclRefExprs`, `LValueToRValue` casts, pointer dereference operator and `MemberExprs`.

Using data flow results to identify output parameters

Let's take a look at how we use data flow analysis to identify an output parameter. The refactoring can be safely done when the data flow algorithm computes a normal state with all of the fields proven to be overwritten in the exit basic block of the function.

```
struct Customer {
  int account_id;
  std::string name;
};

void GetCustomer(Customer* c) {
  // Overwritten: {}
  c->account_id = ...; // Overwritten: {c->account_id}
  if (...) {
    c->name = ...;     // Overwritten: {c->account_id, c->name}
  } else {
    c->name = ...;     // Overwritten: {c->account_id, c->name}
  }
  // Overwritten: {c->account_id, c->name}
}
```

When the data flow algorithm computes a normal state, but not all fields are proven to be overwritten we can't perform the refactoring.

```
void target(bool b, Customer* c) {
  // Overwritten: {}
  if (b) {
    c->account_id = 42;     // Overwritten: {c->account_id}
  } else {
    c->name = "Konrad";  // Overwritten: {c->name}
  }
```

```
  // Overwritten: {}
}
```

Similarly, when the data flow algorithm computes a failure state, we also can't perform the refactoring.

```
Customer* kGlobalCustomer;

void GetCustomer(Customer* c) {
  // Overwritten: {}
  c->account_id = ...;    // Overwritten: {c->account_id}
  if (...) {
    print(c->name);       // Unsafe read
  } else {
    kGlobalCustomer = c;  // Pointer escape
  }
  // Unsafe read, Pointer escape
}
```

Example: finding dead stores

Let's say we want to find redundant stores, because they indicate potential bugs.

```
x = GetX();
x = GetY();
```

The first store to x is never read, probably there is a bug.

The implementation of dead store analysis is very similar to output parameter analysis: we need to track stores and loads, and find stores that were never read.

Liveness analysis is a generalization of this idea, which is often used to answer many related questions, for example:

- finding dead stores,
- finding uninitialized variables,
- finding a good point to deallocate memory,
- finding out if it would be safe to move an object.

Example: definitive initialization

Definitive initialization proves that variables are known to be initialized when read. If we find a variable which is read when not initialized then we generate a warning.

```
void Init() {
  int x;    // x is uninitialized
  if (cond()) {
    x = 10; // x is initialized
  } else {
    x = 20; // x is initialized
  }
  print(x); // x is initialized
}

void Uninit() {
  int x;    // x is uninitialized
  if (cond()) {
    x = 10; // x is initialized
  }
  print(x); // x is maybe uninitialized, x is being read, report a bug.
}
```

For this purpose we can use lattice in a form of a mapping from variable declarations to initialization states; each initialization state is represented by the followingn lattice:

A lattice element could also capture the source locations of the branches that lead us to the corresponding program point. Diagnostics would use this information to show a sample buggy code path to the user.

Example: refactoring raw pointers to `unique_ptr`

Modern idiomatic C++ uses smart pointers to express memory ownership, however in pre-C++11 code one can often find raw pointers that own heap memory blocks.

Imagine that we would like to refactor raw pointers that own memory to `unique_ptr`. There are multiple ways to design a data flow analysis for this problem; let's look at one way to do it.

For example, we would like to refactor the following code that uses raw pointers:

```
void UniqueOwnership1() {
  int *pi = new int;
  if (...) {
    Borrow(pi);
    delete pi;
  } else {
    TakeOwnership(pi);
  }
}
```

into code that uses `unique_ptr`:

```
void UniqueOwnership1() {
  auto pi = std::make_unique<int>();
  if (...) {
    Borrow(pi.get());
  } else {
    TakeOwnership(pi.release());
  }
}
```

This problem can be solved with a lattice in form of map from value declarations to pointer states:

We can perform the refactoring if at the exit of a function `pi` is `Compatible`.

```
void UniqueOwnership1() {
  int *pi;              // pi is Compatible
  pi = new int;         // pi is Defined
  if (...) {
    Borrow(pi);         // pi is Defined
    delete pi;          // pi is Compatible
  } else {
    TakeOwnership(pi); // pi is Compatible
  }
  // pi is Compatible
}
```

Let's look at an example where the raw pointer owns two different memory blocks:

```
void UniqueOwnership2() {
  int *pi = new int;  // pi is Defined
  Borrow(pi);
  delete pi;          // pi is Compatible
  if (smth) {
    pi = new int;     // pi is Defined
    Borrow(pi);
    delete pi;        // pi is Compatible
  }
  // pi is Compatible
}
```

It can be refactored to use `unique_ptr` like this:

```
void UniqueOwnership2() {
  auto pi = make_unique<int>();
  Borrow(pi);
  if (smth) {
    pi = make_unique<int>();
    Borrow(pi);
  }
}
```

In the following example, the raw pointer is used to access the heap object after the ownership has been transferred.

```
void UniqueOwnership3() {
  int *pi = new int; // pi is Defined
  if (...) {
    Borrow(pi);
    delete pi;       // pi is Compatible
  } else {
    vector<unique_ptr<int>> v = {std::unique_ptr(pi)}; // pi is Compatible
    print(*pi);
    use(v);
  }
  // pi is Compatible
}
```

We can refactor this code to use `unique_ptr`, however we would have to introduce a non-owning pointer variable, since we can't use the moved-from `unique_ptr` to access the object:

```
void UniqueOwnership3() {
  std::unique_ptr<int> pi = std::make_unique<int>();
  if (...) {
    Borrow(pi);
  } else {
    int *pi_non_owning = pi.get();
    vector<unique_ptr<int>> v = {std::move(pi)};
    print(*pi_non_owning);
    use(v);
  }
}
```

If the original code didn't call `delete` at the very end of the function, then our refactoring may change the point at which we run the destructor and release memory. Specifically, if there is some user code after `delete`, then extending the lifetime of the object until the end of the function may hold locks for longer than necessary, introduce memory overhead etc.

One solution is to always replace `delete` with a call to `reset()`, and then perform another analysis that removes unnecessary `reset()` calls.

```
void AddedMemoryOverhead() {
  HugeObject *ho = new HugeObject();
  use(ho);
  delete ho; // Release the large amount of memory quickly.
  LongRunningFunction();
}
```

This analysis will refuse to refactor code that mixes borrowed pointer values and unique ownership. In the following code, `GetPtr()` returns a borrowed pointer, which is assigned to `pi`. Then, `pi` is used to hold a uniquely-owned pointer. We don't distinguish between these two assignments, and we want each assignment to be paired with a corresponding sink; otherwise, we transition the pointer to a `Conflicting` state, like in this example.

```
void ConflictingOwnership() {
  int *pi;           // pi is Compatible
  pi = GetPtr();     // pi is Defined
  Borrow(pi);        // pi is Defined
```

```
  pi = new int;      // pi is Conflicting
  Borrow(pi);
  delete pi;
  // pi is Conflicting
}
```

We could still handle this case by finding a maximal range in the code where `pi` could be in the Compatible state, and only refactoring that part.

```
void ConflictingOwnership() {
  int *pi;
  pi = GetPtr();
  Borrow(pi);

  std::unique_ptr<int> pi_unique = std::make_unique<int>();
  Borrow(pi_unique.get());
}
```

Example: finding redundant branch conditions

In the code below `b1` should not be checked in both the outer and inner “if” statements. It is likely there is a bug in this code.

```
int F(bool b1, bool b2) {
  if (b1) {
    f();
    if (b1 && b2) {  // Check `b1` again -- unnecessary!
      g();
    }
  }
}
```

A checker that finds this pattern syntactically is already implemented in ClangTidy using AST matchers (`bugprone-redundant-branch-condition`).

To implement it using the data flow analysis framework, we can produce a warning if any part of the branch condition is implied by the flow condition.

```
int F(bool b1, bool b2) {
  // Flow condition: true.
  if (b1) {
    // Flow condition: b1.
    f();
    if (b1 && b2) { // `b1` is implied by the flow condition.
      g();
    }
  }
}
```

One way to check this implication is to use a SAT solver. Without a SAT solver, we could keep the flow condition in the CNF form and then it would be easy to check the implication.

Example: finding unchecked `std::optional` unwraps

Calling `optional::value()` is only valid if `optional::has_value()` is true. We want to show that when `x.value()` is executed, the flow condition implies `x.has_value()`.

In the example below `x.value()` is accessed safely because it is guarded by the `x.has_value()` check.

```
void Example(std::optional<int> &x) {
  if (x.has_value()) {
    use(x.value());
```

```
  }
}
```

While entering the if branch we deduce that `x.has_value()` is implied by the flow condition.

```
void Example(std::optional<int> x) {
  // Flow condition: true.
  if (x.has_value()) {
    // Flow condition: x.has_value() == true.
    use(x.value());
  }
  // Flow condition: true.
}
```

We also need to prove that `x` is not modified between check and value access. The modification of `x` may be very subtle:

```
void F(std::optional<int> &x);

void Example(std::optional<int> &x) {
  if (x.has_value()) {
    // Flow condition: x.has_value() == true.
    unknown_function(x); // may change x.
    // Flow condition: true.
    use(x.value());
  }
}
```

Example: finding dead code behind A/B experiment flags

Finding dead code is a classic application of data flow analysis.

Unused flags for A/B experiment hide dead code. However, this flavor of dead code is invisible to the compiler because the flag can be turned on at any moment.

We could make a tool that deletes experiment flags. The user tells us which flag they want to delete, and we assume that the it's value is a given constant.

For example, the user could use the tool to remove `example_flag` from this code:

```
DEFINE_FLAG(std::string, example_flag, "", "A sample flag.");

void Example() {
  bool x = GetFlag(FLAGS_example_flag).empty();
  f();
  if (x) {
    g();
  } else {
    h();
  }
}
```

The tool would simplify the code to:

```
void Example() {
  f();
  g();
}
```

We can solve this problem with a classic constant propagation lattice combined with symbolic evaluation.

Example: finding inefficient usages of associative containers

Real-world code often accidentally performs repeated lookups in associative containers:

```
map<int, Employee> xs;
xs[42]->name = "...";
xs[42]->title = "...";
```

To find the above inefficiency we can use the available expressions analysis to understand that `m[42]` is evaluated twice.

```
map<int, Employee> xs;
Employee &e = xs[42];
e->name = "...";
e->title = "...";
```

We can also track the `m.contains()` check in the flow condition to find redundant checks, like in the example below.

```
std::map<int, Employee> xs;
if (!xs.contains(42)) {
  xs.insert({42, someEmployee});
}
```

Example: refactoring types that implicitly convert to each other

Refactoring one strong type to another is difficult, but the compiler can help: once you refactor one reference to the type, the compiler will flag other places where this information flows with type mismatch errors. Unfortunately this strategy does not work when you are refactoring types that implicitly convert to each other, for example, replacing `int32_t` with `int64_t`.

Imagine that we want to change user IDs from 32 to 64-bit integers. In other words, we need to find all integers tainted with user IDs. We can use data flow analysis to implement taint analysis.

```
void UseUser(int32_t user_id) {
  int32_t id = user_id;
  // Variable `id` is tainted with a user ID.
  ...
}
```

Taint analysis is very well suited to this problem because the program rarely branches on user IDs, and almost certainly does not perform any computation (like arithmetic).

AddressSanitizer

Introduction 639
How to build 639
Usage 640
Symbolizing the Reports 640
Additional Checks 641
- Initialization order checking 641
- Stack Use After Return (UAR) 641
- Memory leak detection 641

Issue Suppression 641
- Suppressing Reports in External Libraries 641
- Conditional Compilation with `__has_feature(address_sanitizer)` 642
- Disabling Instrumentation with `__attribute__((no_sanitize("address")))` 642
- Suppressing Errors in Recompiled Code (Ignorelist) 642
- Suppressing memory leaks 642

Code generation control 643
- Instrumentation code outlining 643

Limitations 643
Supported Platforms 643
Current Status 643
More Information 643

Introduction

AddressSanitizer is a fast memory error detector. It consists of a compiler instrumentation module and a run-time library. The tool can detect the following types of bugs:

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- **Use-after-return (clang flag `-fsanitize-address-use-after-return=(never|runtime|always)` default: `runtime`)**
  - Enable with: `ASAN_OPTIONS=detect_stack_use_after_return=1` (already enabled on Linux).
  - Disable with: `ASAN_OPTIONS=detect_stack_use_after_return=0`.
- Use-after-scope (clang flag `-fsanitize-address-use-after-scope`)
- Double-free, invalid free
- Memory leaks (experimental)

Typical slowdown introduced by AddressSanitizer is **2x**.

How to build

Build LLVM/Clang with CMake.

Usage

Simply compile and link your program with `-fsanitize=address` flag. The AddressSanitizer run-time library should be linked to the final executable, so make sure to use `clang` (not `ld`) for the final link step. When linking shared libraries, the AddressSanitizer run-time is not linked, so `-Wl,-z,defs` may cause link errors (don't use it with AddressSanitizer). To get a reasonable performance add `-O1` or higher. To get nicer stack traces in error messages add `-fno-omit-frame-pointer`. To get perfect stack traces you may need to disable inlining (just use `-O1`) and tail call elimination (`-fno-optimize-sibling-calls`).

```
% cat example_UseAfterFree.cc
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];  // BOOM
}

# Compile and link
% clang++ -O1 -g -fsanitize=address -fno-omit-frame-pointer example_UseAfterFree.cc
```

or:

```
# Compile
% clang++ -O1 -g -fsanitize=address -fno-omit-frame-pointer -c example_UseAfterFree.cc
# Link
% clang++ -g -fsanitize=address example_UseAfterFree.o
```

If a bug is detected, the program will print an error message to stderr and exit with a non-zero exit code. AddressSanitizer exits on the first detected error. This is by design:

- This approach allows AddressSanitizer to produce faster and smaller generated code (both by ~5%).
- Fixing bugs becomes unavoidable. AddressSanitizer does not produce false alarms. Once a memory corruption occurs, the program is in an inconsistent state, which could lead to confusing results and potentially misleading subsequent reports.

If your process is sandboxed and you are running on OS X 10.10 or earlier, you will need to set `DYLD_INSERT_LIBRARIES` environment variable and point it to the ASan library that is packaged with the compiler used to build the executable. (You can find the library by searching for dynamic libraries with `asan` in their name.) If the environment variable is not set, the process will try to re-exec. Also keep in mind that when moving the executable to another machine, the ASan library will also need to be copied over.

Symbolizing the Reports

To make AddressSanitizer symbolize its output you need to set the `ASAN_SYMBOLIZER_PATH` environment variable to point to the `llvm-symbolizer` binary (or make sure `llvm-symbolizer` is in your `$PATH`):

```
% ASAN_SYMBOLIZER_PATH=/usr/local/bin/llvm-symbolizer ./a.out
==9442== ERROR: AddressSanitizer heap-use-after-free on address 0x7f7ddab8c084 at pc 0x403c8c bp 0x7fff87fb82d0 sp 0x7fff87fb82c8
READ of size 4 at 0x7f7ddab8c084 thread T0
    #0 0x403c8c in main example_UseAfterFree.cc:4
    #1 0x7f7ddabcac4d in __libc_start_main ??:0
0x7f7ddab8c084 is located 4 bytes inside of 400-byte region [0x7f7ddab8c080,0x7f7ddab8c210)
freed by thread T0 here:
    #0 0x404704 in operator delete[](void*) ??:0
    #1 0x403c53 in main example_UseAfterFree.cc:4
    #2 0x7f7ddabcac4d in __libc_start_main ??:0
previously allocated by thread T0 here:
    #0 0x404544 in operator new[](unsigned long) ??:0
    #1 0x403c43 in main example_UseAfterFree.cc:2
    #2 0x7f7ddabcac4d in __libc_start_main ??:0
==9442== ABORTING
```

If that does not work for you (e.g. your process is sandboxed), you can use a separate script to symbolize the result offline (online symbolization can be force disabled by setting `ASAN_OPTIONS=symbolize=0`):

```
% ASAN_OPTIONS=symbolize=0 ./a.out 2> log
% projects/compiler-rt/lib/asan/scripts/asan_symbolize.py / < log | c++filt
==9442== ERROR: AddressSanitizer heap-use-after-free on address 0x7f7ddab8c084 at pc 0x403c8c bp 0x7fff87fb82d0 sp 0x7fff87fb82c8
READ of size 4 at 0x7f7ddab8c084 thread T0
    #0 0x403c8c in main example_UseAfterFree.cc:4
    #1 0x7f7ddabcac4d in __libc_start_main ??:0
...
```

Note that on macOS you may need to run `dsymutil` on your binary to have the file:line info in the AddressSanitizer reports.

Additional Checks

Initialization order checking

AddressSanitizer can optionally detect dynamic initialization order problems, when initialization of globals defined in one translation unit uses globals defined in another translation unit. To enable this check at runtime, you should set environment variable `ASAN_OPTIONS=check_initialization_order=1`.

Note that this option is not supported on macOS.

Stack Use After Return (UAR)

AddressSanitizer can optionally detect stack use after return problems. This is available by default, or explicitly (`-fsanitize-address-use-after-return=runtime`). To disable this check at runtime, set the environment variable `ASAN_OPTIONS=detect_stack_use_after_return=0`.

Enabling this check (`-fsanitize-address-use-after-return=always`) will reduce code size. The code size may be reduced further by completely eliminating this check (`-fsanitize-address-use-after-return=never`).

To summarize: `-fsanitize-address-use-after-return=<mode>`

- `never`: Completely disables detection of UAR errors (reduces code size).
- `runtime`: Adds the code for detection, but it can be disable via the runtime environment (`ASAN_OPTIONS=detect_stack_use_after_return=0`).
- `always`: Enables detection of UAR errors in all cases. (reduces code size, but not as much as `never`).

Memory leak detection

For more information on leak detector in AddressSanitizer, see LeakSanitizer. The leak detection is turned on by default on Linux, and can be enabled using `ASAN_OPTIONS=detect_leaks=1` on macOS; however, it is not yet supported on other platforms.

Issue Suppression

AddressSanitizer is not expected to produce false positives. If you see one, look again; most likely it is a true positive!

Suppressing Reports in External Libraries

Runtime interposition allows AddressSanitizer to find bugs in code that is not being recompiled. If you run into an issue in external libraries, we recommend immediately reporting it to the library maintainer so that it gets addressed. However, you can use the following suppression mechanism to unblock yourself and continue on with the testing. This suppression mechanism should only be used for suppressing issues in external code; it does not work on code recompiled with AddressSanitizer. To suppress errors in external libraries, set the `ASAN_OPTIONS` environment variable to point to a suppression file. You can either specify the full path to the file or the path of the file relative to the location of your executable.

```
ASAN_OPTIONS=suppressions=MyASan.supp
```

Use the following format to specify the names of the functions or libraries you want to suppress. You can see these in the error report. Remember that the narrower the scope of the suppression, the more bugs you will be able to catch.

```
interceptor_via_fun:NameOfCFunctionToSuppress
interceptor_via_fun:-[ClassName objCMethodToSuppress:]
interceptor_via_lib:NameOfTheLibraryToSuppress
```

Conditional Compilation with `__has_feature(address_sanitizer)`

In some cases one may need to execute different code depending on whether AddressSanitizer is enabled. __has_feature can be used for this purpose.

```
#if defined(__has_feature)
#  if __has_feature(address_sanitizer)
// code that builds only under AddressSanitizer
#  endif
#endif
```

Disabling Instrumentation with `__attribute__((no_sanitize("address")))`

Some code should not be instrumented by AddressSanitizer. One may use the attribute `__attribute__((no_sanitize("address")))` (which has deprecated synonyms *no_sanitize_address* and *no_address_safety_analysis*) to disable instrumentation of a particular function. This attribute may not be supported by other compilers, so we suggest to use it together with `__has_feature(address_sanitizer)`.

The same attribute used on a global variable prevents AddressSanitizer from adding redzones around it and detecting out of bounds accesses.

AddressSanitizer also supports `__attribute__((disable_sanitizer_instrumentation))`. This attribute works similar to `__attribute__((no_sanitize("address")))`, but it also prevents instrumentation performed by other sanitizers.

Suppressing Errors in Recompiled Code (Ignorelist)

AddressSanitizer supports `src` and `fun` entity types in Sanitizer special case list, that can be used to suppress error reports in the specified source files or functions. Additionally, AddressSanitizer introduces `global` and `type` entity types that can be used to suppress error reports for out-of-bound access to globals with certain names and types (you may only specify class or struct types).

You may use an `init` category to suppress reports about initialization-order problems happening in certain source files or with certain global variables.

```
# Suppress error reports for code in a file or in a function:
src:bad_file.cpp
# Ignore all functions with names containing MyFooBar:
fun:*MyFooBar*
# Disable out-of-bound checks for global:
global:bad_array
# Disable out-of-bound checks for global instances of a given class ...
type:Namespace::BadClassName
# ... or a given struct. Use wildcard to deal with anonymous namespace.
type:Namespace2::*::BadStructName
# Disable initialization-order checks for globals:
global:bad_init_global=init
type:*BadInitClassSubstring*=init
src:bad/init/files/*=init
```

Suppressing memory leaks

Memory leak reports produced by LeakSanitizer (if it is run as a part of AddressSanitizer) can be suppressed by a separate file passed as

```
LSAN_OPTIONS=suppressions=MyLSan.supp
```

which contains lines of the form *leak:<pattern>*. Memory leak will be suppressed if pattern matches any function name, source file name, or library name in the symbolized stack trace of the leak report. See full documentation for more details.

Code generation control

Instrumentation code outlining

By default AddressSanitizer inlines the instrumentation code to improve the run-time performance, which leads to increased binary size. Using the (clang flag `-fsanitize-address-outline-instrumentation` default: ``false`) flag forces all code instrumentation to be outlined, which reduces the size of the generated code, but also reduces the run-time performance.

Limitations

- AddressSanitizer uses more real memory than a native run. Exact overhead depends on the allocations sizes. The smaller the allocations you make the bigger the overhead is.
- AddressSanitizer uses more stack memory. We have seen up to 3x increase.
- On 64-bit platforms AddressSanitizer maps (but not reserves) 16+ Terabytes of virtual address space. This means that tools like `ulimit` may not work as usually expected.
- Static linking of executables is not supported.

Supported Platforms

AddressSanitizer is supported on:

- Linux i386/x86_64 (tested on Ubuntu 12.04)
- macOS 10.7 - 10.11 (i386/x86_64)
- iOS Simulator
- Android ARM
- NetBSD i386/x86_64
- FreeBSD i386/x86_64 (tested on FreeBSD 11-current)
- Windows 8.1+ (i386/x86_64)

Ports to various other platforms are in progress.

Current Status

AddressSanitizer is fully functional on supported platforms starting from LLVM 3.1. The test suite is integrated into CMake build and can be run with `make check-asan` command.

The Windows port is functional and is used by Chrome and Firefox, but it is not as well supported as the other ports.

More Information

https://github.com/google/sanitizers/wiki/AddressSanitizer

ThreadSanitizer

Introduction

ThreadSanitizer is a tool that detects data races. It consists of a compiler instrumentation module and a run-time library. Typical slowdown introduced by ThreadSanitizer is about **5x-15x**. Typical memory overhead introduced by ThreadSanitizer is about **5x-10x**.

How to build

Build LLVM/Clang with CMake.

Supported Platforms

ThreadSanitizer is supported on the following OS:

- Android aarch64, x86_64
- Darwin arm64, x86_64
- FreeBSD
- Linux aarch64, x86_64, powerpc64, powerpc64le
- NetBSD

Support for other 64-bit architectures is possible, contributions are welcome. Support for 32-bit platforms is problematic and is not planned.

Usage

Simply compile and link your program with `-fsanitize=thread`. To get a reasonable performance add `-O1` or higher. Use `-g` to get file names and line numbers in the warning messages.

Example:

```
% cat projects/compiler-rt/lib/tsan/lit_tests/tiny_race.c
#include <pthread.h>
int Global;
void *Thread1(void *x) {
  Global = 42;
  return x;
}
int main() {
  pthread_t t;
  pthread_create(&t, NULL, Thread1, NULL);
  Global = 43;
  pthread_join(t, NULL);
  return Global;
}

$ clang -fsanitize=thread -g -O1 tiny_race.c
```

If a bug is detected, the program will print an error message to stderr. Currently, ThreadSanitizer symbolizes its output using an external `addr2line` process (this will be fixed in future).

```
% ./a.out
WARNING: ThreadSanitizer: data race (pid=19219)
  Write of size 4 at 0x7fcf47b21bc0 by thread T1:
    #0 Thread1 tiny_race.c:4 (exe+0x00000000a360)

  Previous write of size 4 at 0x7fcf47b21bc0 by main thread:
    #0 main tiny_race.c:10 (exe+0x00000000a3b4)

  Thread T1 (running) created at:
    #0 pthread_create tsan_interceptors.cc:705 (exe+0x00000000c790)
    #1 main tiny_race.c:9 (exe+0x00000000a3a4)
```

__has_feature(thread_sanitizer)

In some cases one may need to execute different code depending on whether ThreadSanitizer is enabled. __has_feature can be used for this purpose.

```
#if defined(__has_feature)
#  if __has_feature(thread_sanitizer)
// code that builds only under ThreadSanitizer
#  endif
#endif
```

`__attribute__((no_sanitize("thread")))`

Some code should not be instrumented by ThreadSanitizer. One may use the function attribute `no_sanitize("thread")` to disable instrumentation of plain (non-atomic) loads/stores in a particular function. ThreadSanitizer still instruments such functions to avoid false positives and provide meaningful stack traces. This attribute may not be supported by other compilers, so we suggest to use it together with `__has_feature(thread_sanitizer)`.

`__attribute__((disable_sanitizer_instrumentation))`

The `disable_sanitizer_instrumentation` attribute can be applied to functions to prevent all kinds of instrumentation. As a result, it may introduce false positives and incorrect stack traces. Therefore, it should be used with care, and only if absolutely required; for example for certain code that cannot tolerate any instrumentation and resulting side-effects. This attribute overrides `no_sanitize("thread")`.

Ignorelist

ThreadSanitizer supports `src` and `fun` entity types in Sanitizer special case list, that can be used to suppress data race reports in the specified source files or functions. Unlike functions marked with `no_sanitize("thread")` attribute, ignored functions are not instrumented at all. This can lead to false positives due to missed synchronization via atomic operations and missed stack frames in reports.

Limitations

- ThreadSanitizer uses more real memory than a native run. At the default settings the memory overhead is 5x plus 1Mb per each thread. Settings with 3x (less accurate analysis) and 9x (more accurate analysis) overhead are also available.
- ThreadSanitizer maps (but does not reserve) a lot of virtual address space. This means that tools like `ulimit` may not work as usually expected.
- Libc/libstdc++ static linking is not supported.
- Non-position-independent executables are not supported. Therefore, the `fsanitize=thread` flag will cause Clang to act as though the `-fPIE` flag had been supplied if compiling without `-fPIC`, and as though the `-pie` flag had been supplied if linking an executable.

Current Status

ThreadSanitizer is in beta stage. It is known to work on large C++ programs using pthreads, but we do not promise anything (yet). C++11 threading is supported with llvm libc++. The test suite is integrated into CMake build and can be run with `make check-tsan` command.

We are actively working on enhancing the tool — stay tuned. Any help, especially in the form of minimized standalone tests is more than welcome.

More Information

https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual

MemorySanitizer

Introduction 646

How to build 646

Usage 646

`__has_feature(memory_sanitizer)` 647

`__attribute__((no_sanitize("memory")))` 647

`__attribute__((disable_sanitizer_instrumentation))` 647

Ignorelist 647

Report symbolization 647

Origin Tracking 647

Use-after-destruction detection 648

Handling external code 648

Supported Platforms 648

Limitations 648

Current Status 649

More Information 649

Introduction

MemorySanitizer is a detector of uninitialized reads. It consists of a compiler instrumentation module and a run-time library.

Typical slowdown introduced by MemorySanitizer is **3x**.

How to build

Build LLVM/Clang with CMake.

Usage

Simply compile and link your program with `-fsanitize=memory` flag. The MemorySanitizer run-time library should be linked to the final executable, so make sure to use `clang` (not `ld`) for the final link step. When linking shared libraries, the MemorySanitizer run-time is not linked, so `-Wl,-z,defs` may cause link errors (don't use it with MemorySanitizer). To get a reasonable performance add `-O1` or higher. To get meaningful stack traces in error messages add `-fno-omit-frame-pointer`. To get perfect stack traces you may need to disable inlining (just use `-O1`) and tail call elimination (`-fno-optimize-sibling-calls`).

```
% cat umr.cc
#include <stdio.h>

int main(int argc, char** argv) {
  int* a = new int[10];
  a[5] = 0;
  if (a[argc])
    printf("xx\n");
  return 0;
}

% clang -fsanitize=memory -fno-omit-frame-pointer -g -O2 umr.cc
```

If a bug is detected, the program will print an error message to stderr and exit with a non-zero exit code.

```
% ./a.out
WARNING: MemorySanitizer: use-of-uninitialized-value
```

```
    #0 0x7f45944b418a in main umr.cc:6
    #1 0x7f45938b676c in __libc_start_main libc-start.c:226
```

By default, MemorySanitizer exits on the first detected error. If you find the error report hard to understand, try enabling origin tracking.

`__has_feature(memory_sanitizer)`

In some cases one may need to execute different code depending on whether MemorySanitizer is enabled. __has_feature can be used for this purpose.

```
#if defined(__has_feature)
#  if __has_feature(memory_sanitizer)
// code that builds only under MemorySanitizer
#  endif
#endif
```

`__attribute__((no_sanitize("memory")))`

Some code should not be checked by MemorySanitizer. One may use the function attribute `no_sanitize("memory")` to disable uninitialized checks in a particular function. MemorySanitizer may still instrument such functions to avoid false positives. This attribute may not be supported by other compilers, so we suggest to use it together with `__has_feature(memory_sanitizer)`.

`__attribute__((disable_sanitizer_instrumentation))`

The `disable_sanitizer_instrumentation` attribute can be applied to functions to prevent all kinds of instrumentation. As a result, it may introduce false positives and therefore should be used with care, and only if absolutely required; for example for certain code that cannot tolerate any instrumentation and resulting side-effects. This attribute overrides `no_sanitize("memory")`.

Ignorelist

MemorySanitizer supports `src` and `fun` entity types in Sanitizer special case list, that can be used to relax MemorySanitizer checks for certain source files and functions. All “Use of uninitialized value” warnings will be suppressed and all values loaded from memory will be considered fully initialized.

Report symbolization

MemorySanitizer uses an external symbolizer to print files and line numbers in reports. Make sure that `llvm-symbolizer` binary is in `PATH`, or set environment variable `MSAN_SYMBOLIZER_PATH` to point to it.

Origin Tracking

MemorySanitizer can track origins of uninitialized values, similar to Valgrind’s –track-origins option. This feature is enabled by `-fsanitize-memory-track-origins=2` (or simply `-fsanitize-memory-track-origins`) Clang option. With the code from the example above,

```
% cat umr2.cc
#include <stdio.h>

int main(int argc, char** argv) {
  int* a = new int[10];
  a[5] = 0;
  volatile int b = a[argc];
  if (b)
    printf("xx\n");
  return 0;
}

% clang -fsanitize=memory -fsanitize-memory-track-origins=2 -fno-omit-frame-pointer -g -O2 umr2.cc
```

```
% ./a.out
WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x7f7893912f0b in main umr2.cc:7
    #1 0x7f789249b76c in __libc_start_main libc-start.c:226

  Uninitialized value was stored to memory at
    #0 0x7f78938b5c25 in __msan_chain_origin msan.cc:484
    #1 0x7f7893912ecd in main umr2.cc:6

  Uninitialized value was created by a heap allocation
    #0 0x7f7893901cbd in operator new[](unsigned long) msan_new_delete.cc:44
    #1 0x7f7893912e06 in main umr2.cc:4
```

By default, MemorySanitizer collects both allocation points and all intermediate stores the uninitialized value went through. Origin tracking has proved to be very useful for debugging MemorySanitizer reports. It slows down program execution by a factor of 1.5x-2x on top of the usual MemorySanitizer slowdown and increases memory overhead.

Clang option `-fsanitize-memory-track-origins=1` enables a slightly faster mode when MemorySanitizer collects only allocation points but not intermediate stores.

Use-after-destruction detection

MemorySanitizer includes use-after-destruction detection. After invocation of the destructor, the object will be considered no longer readable, and using underlying memory will lead to error reports in runtime. Refer to the standard for lifetime definition.

This feature can be disabled with either:

1. Pass addition Clang option `-fno-sanitize-memory-use-after-dtor` during compilation.
2. Set environment variable *MSAN_OPTIONS=poison_in_dtor=0* before running the program.

Handling external code

MemorySanitizer requires that all program code is instrumented. This also includes any libraries that the program depends on, even libc. Failing to achieve this may result in false reports. For the same reason you may need to replace all inline assembly code that writes to memory with a pure C/C++ code.

Full MemorySanitizer instrumentation is very difficult to achieve. To make it easier, MemorySanitizer runtime library includes 70+ interceptors for the most common libc functions. They make it possible to run MemorySanitizer-instrumented programs linked with uninstrumented libc. For example, the authors were able to bootstrap MemorySanitizer-instrumented Clang compiler by linking it with self-built instrumented libc++ (as a replacement for libstdc++).

Supported Platforms

MemorySanitizer is supported on the following OS:

- Linux
- NetBSD
- FreeBSD

Limitations

- MemorySanitizer uses 2x more real memory than a native run, 3x with origin tracking.
- MemorySanitizer maps (but not reserves) 64 Terabytes of virtual address space. This means that tools like `ulimit` may not work as usually expected.
- Static linking is not supported.

- Older versions of MSan (LLVM 3.7 and older) didn't work with non-position-independent executables, and could fail on some Linux kernel versions with disabled ASLR. Refer to documentation for older versions for more details.
- MemorySanitizer might be incompatible with position-independent executables from FreeBSD 13 but there is a check done at runtime and throws a warning in this case.

Current Status

MemorySanitizer is known to work on large real-world programs (like Clang/LLVM itself) that can be recompiled from source, including all dependent libraries.

More Information

https://github.com/google/sanitizers/wiki/MemorySanitizer

UndefinedBehaviorSanitizer

Introduction 649
How to build 650
Usage 650
Available checks 650
Volatile 652
Minimal Runtime 652
Stack traces and report symbolization 652
Logging 652
Silencing Unsigned Integer Overflow 653
Issue Suppression 653
Disabling Instrumentation with `__attribute__((no_sanitize("undefined")))` 653
Suppressing Errors in Recompiled Code (Ignorelist) 653
Runtime suppressions 653
Supported Platforms 653
Current Status 654
Additional Configuration 654
Example 654
More Information 654

Introduction

UndefinedBehaviorSanitizer (UBSan) is a fast undefined behavior detector. UBSan modifies the program at compile-time to catch various kinds of undefined behavior during program execution, for example:

- Array subscript out of bounds, where the bounds can be statically determined
- Bitwise shifts that are out of bounds for their data type
- Dereferencing misaligned or null pointers
- Signed integer overflow
- Conversion to, from, or between floating-point types which would overflow the destination

See the full list of available checks below.

UBSan has an optional run-time library which provides better error reporting. The checks have small runtime cost and no impact on address space layout or ABI.

How to build

Build LLVM/Clang with CMake.

Usage

Use `clang++` to compile and link your program with `-fsanitize=undefined` flag. Make sure to use `clang++` (not `ld`) as a linker, so that your executable is linked with proper UBSan runtime libraries. You can use `clang` instead of `clang++` if you're compiling/linking C code.

```
% cat test.cc
int main(int argc, char **argv) {
  int k = 0x7fffffff;
  k += argc;
  return 0;
}
% clang++ -fsanitize=undefined test.cc
% ./a.out
test.cc:3:5: runtime error: signed integer overflow: 2147483647 + 1 cannot be represented in type 'int'
```

You can enable only a subset of checks offered by UBSan, and define the desired behavior for each kind of check:

- `-fsanitize=...`: print a verbose error report and continue execution (default);
- `-fno-sanitize-recover=...`: print a verbose error report and exit the program;
- `-fsanitize-trap=...`: execute a trap instruction (doesn't require UBSan run-time support).
- `-fno-sanitize=...`: disable any check, e.g., -fno-sanitize=alignment.

Note that the `trap` / `recover` options do not enable the corresponding sanitizer, and in general need to be accompanied by a suitable `-fsanitize=` flag.

For example if you compile/link your program as:

```
% clang++ -fsanitize=signed-integer-overflow,null,alignment -fno-sanitize-recover=null -fsanitize-trap=alignment
```

the program will continue execution after signed integer overflows, exit after the first invalid use of a null pointer, and trap after the first use of misaligned pointer.

Available checks

Available checks are:

- `-fsanitize=alignment`: Use of a misaligned pointer or creation of a misaligned reference. Also sanitizes assume_aligned-like attributes.
- `-fsanitize=bool`: Load of a `bool` value which is neither `true` nor `false`.
- `-fsanitize=builtin`: Passing invalid values to compiler builtins.
- `-fsanitize=bounds`: Out of bounds array indexing, in cases where the array bound can be statically determined. The check includes `-fsanitize=array-bounds` and `-fsanitize=local-bounds`. Note that `-fsanitize=local-bounds` is not included in `-fsanitize=undefined`.
- `-fsanitize=enum`: Load of a value of an enumerated type which is not in the range of representable values for that enumerated type.
- `-fsanitize=float-cast-overflow`: Conversion to, from, or between floating-point types which would overflow the destination. Because the range of representable values for all floating-point types supported by Clang is [-inf, +inf], the only cases detected are conversions from floating point to integer types.
- `-fsanitize=float-divide-by-zero`: Floating point division by zero. This is undefined per the C and C++ standards, but is defined by Clang (and by ISO/IEC/IEEE 60559 / IEEE 754) as producing either an infinity or NaN value, so is not included in `-fsanitize=undefined`.
- `-fsanitize=function`: Indirect call of a function through a function pointer of the wrong type (Darwin/Linux, C++ and x86/x86_64 only).

- `-fsanitize=implicit-unsigned-integer-truncation`, `-fsanitize=implicit-signed-integer-truncation`: Implicit conversion from integer of larger bit width to smaller bit width, if that results in data loss. That is, if the demoted value, after casting back to the original width, is not equal to the original value before the downcast. The `-fsanitize=implicit-unsigned-integer-truncation` handles conversions between two `unsigned` types, while `-fsanitize=implicit-signed-integer-truncation` handles the rest of the conversions - when either one, or both of the types are signed. Issues caught by these sanitizers are not undefined behavior, but are often unintentional.
- `-fsanitize=implicit-integer-sign-change`: Implicit conversion between integer types, if that changes the sign of the value. That is, if the original value was negative and the new value is positive (or zero), or the original value was positive, and the new value is negative. Issues caught by this sanitizer are not undefined behavior, but are often unintentional.
- `-fsanitize=integer-divide-by-zero`: Integer division by zero.
- `-fsanitize=nonnull-attribute`: Passing null pointer as a function parameter which is declared to never be null.
- `-fsanitize=null`: Use of a null pointer or creation of a null reference.
- `-fsanitize=nullability-arg`: Passing null as a function parameter which is annotated with `_Nonnull`.
- `-fsanitize=nullability-assign`: Assigning null to an lvalue which is annotated with `_Nonnull`.
- `-fsanitize=nullability-return`: Returning null from a function with a return type annotated with `_Nonnull`.
- `-fsanitize=objc-cast`: Invalid implicit cast of an ObjC object pointer to an incompatible type. This is often unintentional, but is not undefined behavior, therefore the check is not a part of the `undefined` group. Currently only supported on Darwin.
- `-fsanitize=object-size`: An attempt to potentially use bytes which the optimizer can determine are not part of the object being accessed. This will also detect some types of undefined behavior that may not directly access memory, but are provably incorrect given the size of the objects involved, such as invalid downcasts and calling methods on invalid pointers. These checks are made in terms of `__builtin_object_size`, and consequently may be able to detect more problems at higher optimization levels.
- `-fsanitize=pointer-overflow`: Performing pointer arithmetic which overflows, or where either the old or new pointer value is a null pointer (or in C, when they both are).
- `-fsanitize=return`: In C++, reaching the end of a value-returning function without returning a value.
- `-fsanitize=returns-nonnull-attribute`: Returning null pointer from a function which is declared to never return null.
- `-fsanitize=shift`: Shift operators where the amount shifted is greater or equal to the promoted bit-width of the left hand side or less than zero, or where the left hand side is negative. For a signed left shift, also checks for signed overflow in C, and for unsigned overflow in C++. You can use `-fsanitize=shift-base` or `-fsanitize=shift-exponent` to check only left-hand side or right-hand side of shift operation, respectively.
- `-fsanitize=unsigned-shift-base`: check that an unsigned left-hand side of a left shift operation doesn't overflow.
- `-fsanitize=signed-integer-overflow`: Signed integer overflow, where the result of a signed integer computation cannot be represented in its type. This includes all the checks covered by `-ftrapv`, as well as checks for signed division overflow (`INT_MIN/-1`), but not checks for lossy implicit conversions performed before the computation (see `-fsanitize=implicit-conversion`). Both of these two issues are handled by `-fsanitize=implicit-conversion` group of checks.
- `-fsanitize=unreachable`: If control flow reaches an unreachable program point.
- `-fsanitize=unsigned-integer-overflow`: Unsigned integer overflow, where the result of an unsigned integer computation cannot be represented in its type. Unlike signed integer overflow, this is not undefined behavior, but it is often unintentional. This sanitizer does not check for lossy implicit conversions performed before such a computation (see `-fsanitize=implicit-conversion`).

- `-fsanitize=vla-bound`: A variable-length array whose bound does not evaluate to a positive value.
- `-fsanitize=vptr`: Use of an object whose vptr indicates that it is of the wrong dynamic type, or that its lifetime has not begun or has ended. Incompatible with `-fno-rtti`. Link must be performed by `clang++`, not `clang`, to make sure C++-specific parts of the runtime library and C++ standard libraries are present.

You can also use the following check groups:

- `-fsanitize=undefined`: All of the checks listed above other than `float-divide-by-zero`, `unsigned-integer-overflow`, `implicit-conversion`, `local-bounds` and the `nullability-*` group of checks.
- `-fsanitize=undefined-trap`: Deprecated alias of `-fsanitize=undefined`.
- `-fsanitize=implicit-integer-truncation`: Catches lossy integral conversions. Enables `implicit-signed-integer-truncation` and `implicit-unsigned-integer-truncation`.
- `-fsanitize=implicit-integer-arithmetic-value-change`: Catches implicit conversions that change the arithmetic value of the integer. Enables `implicit-signed-integer-truncation` and `implicit-integer-sign-change`.
- `-fsanitize=implicit-conversion`: Checks for suspicious behavior of implicit conversions. Enables `implicit-unsigned-integer-truncation`, `implicit-signed-integer-truncation`, and `implicit-integer-sign-change`.
- `-fsanitize=integer`: Checks for undefined or suspicious integer behavior (e.g. unsigned integer overflow). Enables `signed-integer-overflow`, `unsigned-integer-overflow`, `shift`, `integer-divide-by-zero`, `implicit-unsigned-integer-truncation`, `implicit-signed-integer-truncation`, and `implicit-integer-sign-change`.
- `-fsanitize=nullability`: Enables `nullability-arg`, `nullability-assign`, and `nullability-return`. While violating nullability does not have undefined behavior, it is often unintentional, so UBSan offers to catch it.

Volatile

The `null`, `alignment`, `object-size`, `local-bounds`, and `vptr` checks do not apply to pointers to types with the `volatile` qualifier.

Minimal Runtime

There is a minimal UBSan runtime available suitable for use in production environments. This runtime has a small attack surface. It only provides very basic issue logging and deduplication, and does not support `-fsanitize=function` and `-fsanitize=vptr` checking.

To use the minimal runtime, add `-fsanitize-minimal-runtime` to the clang command line options. For example, if you're used to compiling with `-fsanitize=undefined`, you could enable the minimal runtime with `-fsanitize=undefined -fsanitize-minimal-runtime`.

Stack traces and report symbolization

If you want UBSan to print symbolized stack trace for each error report, you will need to:

1. Compile with `-g` and `-fno-omit-frame-pointer` to get proper debug information in your binary.
2. Run your program with environment variable `UBSAN_OPTIONS=print_stacktrace=1`.
3. Make sure `llvm-symbolizer` binary is in `PATH`.

Logging

The default log file for diagnostics is "stderr". To log diagnostics to another file, you can set `UBSAN_OPTIONS=log_path=...`.

Silencing Unsigned Integer Overflow

To silence reports from unsigned integer overflow, you can set `UBSAN_OPTIONS=silence_unsigned_overflow=1`. This feature, combined with `-fsanitize-recover=unsigned-integer-overflow`, is particularly useful for providing fuzzing signal without blowing up logs.

Issue Suppression

UndefinedBehaviorSanitizer is not expected to produce false positives. If you see one, look again; most likely it is a true positive!

Disabling Instrumentation with `__attribute__((no_sanitize("undefined")))`

You disable UBSan checks for particular functions with `__attribute__((no_sanitize("undefined")))`. You can use all values of `-fsanitize=` flag in this attribute, e.g. if your function deliberately contains possible signed integer overflow, you can use `__attribute__((no_sanitize("signed-integer-overflow")))`.

This attribute may not be supported by other compilers, so consider using it together with `#if defined(__clang__)`.

Suppressing Errors in Recompiled Code (Ignorelist)

UndefinedBehaviorSanitizer supports `src` and `fun` entity types in Sanitizer special case list, that can be used to suppress error reports in the specified source files or functions.

Runtime suppressions

Sometimes you can suppress UBSan error reports for specific files, functions, or libraries without recompiling the code. You need to pass a path to suppression file in a `UBSAN_OPTIONS` environment variable.

```
UBSAN_OPTIONS=suppressions=MyUBSan.supp
```

You need to specify a check you are suppressing and the bug location. For example:

```
signed-integer-overflow:file-with-known-overflow.cpp
alignment:function_doing_unaligned_access
vptr:shared_object_with_vptr_failures.so
```

There are several limitations:

- Sometimes your binary must have enough debug info and/or symbol table, so that the runtime could figure out source file or function name to match against the suppression.
- It is only possible to suppress recoverable checks. For the example above, you can additionally pass `-fsanitize-recover=signed-integer-overflow,alignment,vptr`, although most of UBSan checks are recoverable by default.
- Check groups (like `undefined`) can't be used in suppressions file, only fine-grained checks are supported.

Supported Platforms

UndefinedBehaviorSanitizer is supported on the following operating systems:

- Android
- Linux
- NetBSD
- FreeBSD
- OpenBSD
- macOS
- Windows

The runtime library is relatively portable and platform independent. If the OS you need is not listed above, UndefinedBehaviorSanitizer may already work for it, or could be made to work with a minor porting effort.

Current Status

UndefinedBehaviorSanitizer is available on selected platforms starting from LLVM 3.3. The test suite is integrated into the CMake build and can be run with `check-ubsan` command.

Additional Configuration

UndefinedBehaviorSanitizer adds static check data for each check unless it is in trap mode. This check data includes the full file name. The option `-fsanitize-undefined-strip-path-components=N` can be used to trim this information. If `N` is positive, file information emitted by UndefinedBehaviorSanitizer will drop the first `N` components from the file path. If `N` is negative, the last `N` components will be kept.

Example

For a file called `/code/library/file.cpp`, here is what would be emitted:

- Default (No flag, or `-fsanitize-undefined-strip-path-components=0`): `/code/library/file.cpp`
- `-fsanitize-undefined-strip-path-components=1`: `code/library/file.cpp`
- `-fsanitize-undefined-strip-path-components=2`: `library/file.cpp`
- `-fsanitize-undefined-strip-path-components=-1`: `file.cpp`
- `-fsanitize-undefined-strip-path-components=-2`: `library/file.cpp`

More Information

- From Oracle blog, including a discussion of error messages: Improving Application Security with UndefinedBehaviorSanitizer (UBSan) and GCC
- From LLVM project blog: What Every C Programmer Should Know About Undefined Behavior
- From John Regehr's *Embedded in Academia* blog: A Guide to Undefined Behavior in C and C++

DataFlowSanitizer

DataFlowSanitizer Design Document

This document sets out the design for DataFlowSanitizer, a general dynamic data flow analysis. Unlike other Sanitizer tools, this tool is not designed to detect a specific class of bugs on its own. Instead, it provides a generic dynamic data flow analysis framework to be used by clients to help detect application-specific issues within their own code.

DataFlowSanitizer is a program instrumentation which can associate a number of taint labels with any data stored in any memory region accessible by the program. The analysis is dynamic, which means that it operates on a running program, and tracks how the labels propagate through that program.

Use Cases

This instrumentation can be used as a tool to help monitor how data flows from a program's inputs (sources) to its outputs (sinks). This has applications from a privacy/security perspective in that one can audit how a sensitive data item is used within a program and ensure it isn't exiting the program anywhere it shouldn't be.

Interface

A number of functions are provided which will attach taint labels to memory regions and extract the set of labels associated with a specific memory region. These functions are declared in the header file `sanitizer/dfsan_interface.h`.

```
/// Sets the label for each address in [addr,addr+size) to \c label.
void dfsan_set_label(dfsan_label label, void *addr, size_t size);

/// Sets the label for each address in [addr,addr+size) to the union of the
/// current label for that address and \c label.
void dfsan_add_label(dfsan_label label, void *addr, size_t size);

/// Retrieves the label associated with the given data.
///
/// The type of 'data' is arbitrary.  The function accepts a value of any type,
/// which can be truncated or extended (implicitly or explicitly) as necessary.
/// The truncation/extension operations will preserve the label of the original
/// value.
dfsan_label dfsan_get_label(long data);

/// Retrieves the label associated with the data at the given address.
dfsan_label dfsan_read_label(const void *addr, size_t size);

/// Returns whether the given label label contains the label elem.
int dfsan_has_label(dfsan_label label, dfsan_label elem);

/// Computes the union of \c l1 and \c l2, resulting in a union label.
dfsan_label dfsan_union(dfsan_label l1, dfsan_label l2);

/// Flushes the DFSan shadow, i.e. forgets about all labels currently associated
/// with the application memory.  Use this call to start over the taint tracking
/// within the same process.
///
/// Note: If another thread is working with tainted data during the flush, that
/// taint could still be written to shadow after the flush.
void dfsan_flush(void);
```

The following functions are provided to check origin tracking status and results.

```
/// Retrieves the immediate origin associated with the given data. The returned
/// origin may point to another origin.
///
/// The type of 'data' is arbitrary. The function accepts a value of any type,
/// which can be truncated or extended (implicitly or explicitly) as necessary.
/// The truncation/extension operations will preserve the label of the original
/// value.
dfsan_origin dfsan_get_origin(long data);

/// Retrieves the very first origin associated with the data at the given
/// address.
dfsan_origin dfsan_get_init_origin(const void *addr);

/// Prints the origin trace of the label at the address `addr` to stderr. It also
/// prints description at the beginning of the trace. If origin tracking is not
/// on, or the address is not labeled, it prints nothing.
void dfsan_print_origin_trace(const void *addr, const char *description);

/// Prints the origin trace of the label at the address `addr` to a pre-allocated
/// output buffer. If origin tracking is not on, or the address is`
/// not labeled, it prints nothing.
///
/// `addr` is the tainted memory address whose origin we are printing.
/// `description` is a description printed at the beginning of the trace.
/// `out_buf` is the output buffer to write the results to. `out_buf_size` is
/// the size of `out_buf`. The function returns the number of symbols that
/// should have been written to `out_buf` (not including trailing null byte '\0').
```

```
/// Thus, the string is truncated iff return value is not less than `out_buf_size`.
size_t dfsan_sprint_origin_trace(const void *addr, const char *description,
                                 char *out_buf, size_t out_buf_size);

/// Returns the value of `-dfsan-track-origins`.
int dfsan_get_track_origins(void);
```

The following functions are provided to register hooks called by custom wrappers.

```
/// Sets a callback to be invoked on calls to `write`.  The callback is invoked
/// before the write is done. The write is not guaranteed to succeed when the
/// callback executes. Pass in NULL to remove any callback.
typedef void (*dfsan_write_callback_t)(int fd, const void *buf, size_t count);
void dfsan_set_write_callback(dfsan_write_callback_t labeled_write_callback);

/// Callbacks to be invoked on calls to `memcmp` or `strncmp`.
void dfsan_weak_hook_memcmp(void *caller_pc, const void *s1, const void *s2,
                            size_t n, dfsan_label s1_label,
                            dfsan_label s2_label, dfsan_label n_label);
void dfsan_weak_hook_strncmp(void *caller_pc, const char *s1, const char *s2,
                            size_t n, dfsan_label s1_label,
                            dfsan_label s2_label, dfsan_label n_label);
```

Taint label representation

We use an 8-bit unsigned integer for the representation of a label. The label identifier 0 is special, and means that the data item is unlabelled. This is optimizing for low CPU and code size overhead of the instrumentation. When a label union operation is requested at a join point (any arithmetic or logical operation with two or more operands, such as addition), we can simply OR the two labels in O(1).

Users are responsible for managing the 8 integer labels (i.e., keeping track of what labels they have used so far, picking one that is yet unused, etc).

Origin tracking trace representation

An origin tracking trace is a list of chains. Each chain has a stack trace where the DFSan runtime records a label propagation, and a pointer to its previous chain. The very first chain does not point to any chain.

Every four 4-bytes aligned application bytes share a 4-byte origin trace ID. A 4-byte origin trace ID contains a 4-bit depth and a 28-bit hash ID of a chain.

A chain ID is calculated as a hash from a chain structure. A chain structure contains a stack ID and the previous chain ID. The chain head has a zero previous chain ID. A stack ID is a hash from a stack trace. The 4-bit depth limits the maximal length of a path. The environment variable `origin_history_size` can set the depth limit. Non-positive values mean unlimited. Its default value is 16. When reaching the limit, origin tracking ignores following propagation chains.

The first chain of a trace starts by *dfsan_set_label* with non-zero labels. A new chain is appended at the end of a trace at stores or memory transfers when `-dfsan-track-origins` is 1. Memory transfers include LLVM memory transfer instructions, glibc memcpy and memmove. When `-dfsan-track-origins` is 2, a new chain is also appended at loads.

Other instructions do not create new chains, but simply propagate origin trace IDs. If an instruction has more than one operands with non-zero labels, the origin treace ID of the last operand with non-zero label is propagated to the result of the instruction.

Memory layout and label management

The following is the memory layout for Linux/x86_64:

Start	End	Use
0x700000000000	0x800000000000	application 3

Start	End	Use
0x610000000000	0x700000000000	unused
0x600000000000	0x610000000000	origin 1
0x510000000000	0x600000000000	application 2
0x500000000000	0x510000000000	shadow 1
0x400000000000	0x500000000000	unused
0x300000000000	0x400000000000	origin 3
0x200000000000	0x300000000000	shadow 3
0x110000000000	0x200000000000	origin 2
0x100000000000	0x110000000000	unused
0x010000000000	0x100000000000	shadow 2
0x000000000000	0x010000000000	application 1

Each byte of application memory corresponds to a single byte of shadow memory, which is used to store its taint label. We map memory, shadow, and origin regions to each other with these masks and offsets:

- shadow_addr = memory_addr ^ 0x500000000000
- origin_addr = shadow_addr + 0x100000000000

As for LLVM SSA registers, we have not found it necessary to associate a label with each byte or bit of data, as some other tools do. Instead, labels are associated directly with registers. Loads will result in a union of all shadow labels corresponding to bytes loaded, and stores will result in a copy of the label of the stored value to the shadow of all bytes stored to.

Propagating labels through arguments

In order to propagate labels through function arguments and return values, DataFlowSanitizer changes the ABI of each function in the translation unit. There are currently two supported ABIs:

- Args – Argument and return value labels are passed through additional arguments and by modifying the return type.
- TLS – Argument and return value labels are passed through TLS variables `__dfsan_arg_tls` and `__dfsan_retval_tls`.

The main advantage of the TLS ABI is that it is more tolerant of ABI mismatches (TLS storage is not shared with any other form of storage, whereas extra arguments may be stored in registers which under the native ABI are not used for parameter passing and thus could contain arbitrary values). On the other hand the args ABI is more efficient and allows ABI mismatches to be more easily identified by checking for nonzero labels in nominally unlabelled programs.

Implementing the ABI list

The ABI list provides a list of functions which conform to the native ABI, each of which is callable from an instrumented program. This is implemented by replacing each reference to a native ABI function with a reference to a function which uses the instrumented ABI. Such functions are automatically-generated wrappers for the native functions. For example, given the ABI list example provided in the user manual, the following wrappers will be generated under the args ABI:

```
define linkonce_odr { i8*, i16 } @"dfsw$malloc"(i64 %0, i16 %1) {
entry:
  %2 = call i8* @malloc(i64 %0)
  %3 = insertvalue { i8*, i16 } undef, i8* %2, 0
  %4 = insertvalue { i8*, i16 } %3, i16 0, 1
  ret { i8*, i16 } %4
}

define linkonce_odr { i32, i16 } @"dfsw$tolower"(i32 %0, i16 %1) {
entry:
```

```
  %2 = call i32 @tolower(i32 %0)
  %3 = insertvalue { i32, i16 } undef, i32 %2, 0
  %4 = insertvalue { i32, i16 } %3, i16 %1, 1
  ret { i32, i16 } %4
}

define linkonce_odr { i8*, i16 } @"dfsw$memcpy"(i8* %0, i8* %1, i64 %2, i16 %3, i16 %4, i16 %5) {
entry:
  %labelreturn = alloca i16
  %6 = call i8* @__dfsw_memcpy(i8* %0, i8* %1, i64 %2, i16 %3, i16 %4, i16 %5, i16* %labelreturn)
  %7 = load i16* %labelreturn
  %8 = insertvalue { i8*, i16 } undef, i8* %6, 0
  %9 = insertvalue { i8*, i16 } %8, i16 %7, 1
  ret { i8*, i16 } %9
}
```

As an optimization, direct calls to native ABI functions will call the native ABI function directly and the pass will compute the appropriate label internally. This has the advantage of reducing the number of union operations required when the return value label is known to be zero (i.e. `discard` functions, or `functional` functions with known unlabelled arguments).

Checking ABI Consistency

DFSan changes the ABI of each function in the module. This makes it possible for a function with the native ABI to be called with the instrumented ABI, or vice versa, thus possibly invoking undefined behavior. A simple way of statically detecting instances of this problem is to append the suffix ".dfsan" to the name of each instrumented-ABI function.

This will not catch every such problem; in particular function pointers passed across the instrumented-native barrier cannot be used on the other side. These problems could potentially be caught dynamically.

Introduction 658

How to build libc++ with DFSan 658

Usage 659

- ABI List 659
- Compilation Flags 660
- Environment Variables 661

Example 661

Origin Tracking 662

Current status 663

Design 663

Introduction

DataFlowSanitizer is a generalised dynamic data flow analysis.

Unlike other Sanitizer tools, this tool is not designed to detect a specific class of bugs on its own. Instead, it provides a generic dynamic data flow analysis framework to be used by clients to help detect application-specific issues within their own code.

How to build libc++ with DFSan

DFSan requires either all of your code to be instrumented or for uninstrumented functions to be listed as `uninstrumented` in the ABI list.

If you'd like to have instrumented libc++ functions, then you need to build it with DFSan instrumentation from source. Here is an example of how to build libc++ and the libc++ ABI with data flow sanitizer instrumentation.

```
mkdir libcxx-build
cd libcxx-build
```

```
# An example using ninja
cmake -GNinja -S <monorepo-root>/runtimes \
  -DCMAKE_C_COMPILER=clang \
  -DCMAKE_CXX_COMPILER=clang++ \
  -DLLVM_USE_SANITIZER="DataFlow" \
  -DLLVM_ENABLE_RUNTIMES="libcxx;libcxxabi"

ninja cxx cxxabi
```

Note: Ensure you are building with a sufficiently new version of Clang.

Usage

With no program changes, applying DataFlowSanitizer to a program will not alter its behavior. To use DataFlowSanitizer, the program uses API functions to apply tags to data to cause it to be tracked, and to check the tag of a specific data item. DataFlowSanitizer manages the propagation of tags through the program according to its data flow.

The APIs are defined in the header file `sanitizer/dfsan_interface.h`. For further information about each function, please refer to the header file.

ABI List

DataFlowSanitizer uses a list of functions known as an ABI list to decide whether a call to a specific function should use the operating system's native ABI or whether it should use a variant of this ABI that also propagates labels through function parameters and return values. The ABI list file also controls how labels are propagated in the former case. DataFlowSanitizer comes with a default ABI list which is intended to eventually cover the glibc library on Linux but it may become necessary for users to extend the ABI list in cases where a particular library or function cannot be instrumented (e.g. because it is implemented in assembly or another language which DataFlowSanitizer does not support) or a function is called from a library or function which cannot be instrumented.

DataFlowSanitizer's ABI list file is a Sanitizer special case list. The pass treats every function in the `uninstrumented` category in the ABI list file as conforming to the native ABI. Unless the ABI list contains additional categories for those functions, a call to one of those functions will produce a warning message, as the labelling behavior of the function is unknown. The other supported categories are `discard`, `functional` and `custom`.

- `discard` – To the extent that this function writes to (user-accessible) memory, it also updates labels in shadow memory (this condition is trivially satisfied for functions which do not write to user-accessible memory). Its return value is unlabelled.
- `functional` – Like `discard`, except that the label of its return value is the union of the label of its arguments.
- `custom` – Instead of calling the function, a custom wrapper `__dfsw_F` is called, where `F` is the name of the function. This function may wrap the original function or provide its own implementation. This category is generally used for uninstrumentable functions which write to user-accessible memory or which have more complex label propagation behavior. The signature of `__dfsw_F` is based on that of `F` with each argument having a label of type `dfsan_label` appended to the argument list. If `F` is of non-void return type a final argument of type `dfsan_label *` is appended to which the custom function can store the label for the return value. For example:

```
void f(int x);
void __dfsw_f(int x, dfsan_label x_label);

void *memcpy(void *dest, const void *src, size_t n);
void *__dfsw_memcpy(void *dest, const void *src, size_t n,
                    dfsan_label dest_label, dfsan_label src_label,
                    dfsan_label n_label, dfsan_label *ret_label);
```

If a function defined in the translation unit being compiled belongs to the `uninstrumented` category, it will be compiled so as to conform to the native ABI. Its arguments will be assumed to be unlabelled, but it will propagate labels in shadow memory.

For example:

```
# main is called by the C runtime using the native ABI.
fun:main=uninstrumented
fun:main=discard

# malloc only writes to its internal data structures, not user-accessible memory.
fun:malloc=uninstrumented
fun:malloc=discard

# tolower is a pure function.
fun:tolower=uninstrumented
fun:tolower=functional

# memcpy needs to copy the shadow from the source to the destination region.
# This is done in a custom function.
fun:memcpy=uninstrumented
fun:memcpy=custom
```

For instrumented functions, the ABI list supports a `force_zero_labels` category, which will make all stores and return values set zero labels. Functions should never be labelled with both `force_zero_labels` and `uninstrumented` or any of the unistrumented wrapper kinds.

For example:

```
# e.g. void writes_data(char* out_buf, int out_buf_len) {...}
# Applying force_zero_labels will force out_buf shadow to zero.
fun:writes_data=force_zero_labels
```

Compilation Flags

- `-dfsan-abilist` – The additional ABI list files that control how shadow parameters are passed. File names are separated by comma.
- `-dfsan-combine-pointer-labels-on-load` – Controls whether to include or ignore the labels of pointers in load instructions. Its default value is true. For example:

```
v = *p;
```

If the flag is true, the label of `v` is the union of the label of `p` and the label of `*p`. If the flag is false, the label of `v` is the label of just `*p`.

- `-dfsan-combine-pointer-labels-on-store` – Controls whether to include or ignore the labels of pointers in store instructions. Its default value is false. For example:

```
*p = v;
```

If the flag is true, the label of `*p` is the union of the label of `p` and the label of `v`. If the flag is false, the label of `*p` is the label of just `v`.

- `-dfsan-combine-offset-labels-on-gep` – Controls whether to propagate labels of offsets in GEP instructions. Its default value is true. For example:

```
p += i;
```

If the flag is true, the label of `p` is the union of the label of `p` and the label of `i`. If the flag is false, the label of `p` is unchanged.

- `-dfsan-track-select-control-flow` – Controls whether to track the control flow of select instructions. Its default value is true. For example:

```
v = b? v1: v2;
```

If the flag is true, the label of `v` is the union of the labels of `b`, `v1` and `v2`. If the flag is false, the label of `v` is the union of the labels of just `v1` and `v2`.

- `-dfsan-event-callbacks` – An experimental feature that inserts callbacks for certain data events. Currently callbacks are only inserted for loads, stores, memory transfers (i.e. memcpy and memmove), and

comparisons. Its default value is false. If this flag is set to true, a user must provide definitions for the following callback functions:

```
void __dfsan_load_callback(dfsan_label Label, void* Addr);
void __dfsan_store_callback(dfsan_label Label, void* Addr);
void __dfsan_mem_transfer_callback(dfsan_label *Start, size_t Len);
void __dfsan_cmp_callback(dfsan_label CombinedLabel);
```

- `-dfsan-conditional-callbacks` – An experimental feature that inserts callbacks for control flow conditional expressions. This can be used to find where tainted values can control execution.

  In addition to this compilation flag, a callback handler must be registered using `dfsan_set_conditional_callback(my_callback);`, where my_callback is a function with a signature matching `void my_callback(dfsan_label l, dfsan_origin o);`. This signature is the same when origin tracking is disabled - in this case the dfsan_origin passed in it will always be 0.

  The callback will only be called when a tainted value reaches a conditional expression for control flow (such as an if's condition). The callback will be skipped for conditional expressions inside signal handlers, as this is prone to deadlock. Tainted values used in conditional expressions inside signal handlers will instead be aggregated via bitwise or, and can be accessed using `dfsan_label dfsan_get_labels_in_signal_conditional();`.
- `-dfsan-track-origins` – Controls how to track origins. When its value is 0, the runtime does not track origins. When its value is 1, the runtime tracks origins at memory store operations. When its value is 2, the runtime tracks origins at memory load and store operations. Its default value is 0.
- `-dfsan-instrument-with-call-threshold` – If a function being instrumented requires more than this number of origin stores, use callbacks instead of inline checks (-1 means never use callbacks). Its default value is 3500.

Environment Variables

- `warn_unimplemented` – Whether to warn on unimplemented functions. Its default value is false.
- `strict_data_dependencies` – Whether to propagate labels only when there is explicit obvious data dependency (e.g., when comparing strings, ignore the fact that the output of the comparison might be implicit data-dependent on the content of the strings). This applies only to functions with `custom` category in ABI list. Its default value is true.
- `origin_history_size` – The limit of origin chain length. Non-positive values mean unlimited. Its default value is 16.
- `origin_history_per_stack_limit` – The limit of origin node's references count. Non-positive values mean unlimited. Its default value is 20000.
- `store_context_size` – The depth limit of origin tracking stack traces. Its default value is 20.
- `zero_in_malloc` – Whether to zero shadow space of new allocated memory. Its default value is true.
- `zero_in_free` — Whether to zero shadow space of deallocated memory. Its default value is true.

Example

DataFlowSanitizer supports up to 8 labels, to achieve low CPU and code size overhead. Base labels are simply 8-bit unsigned integers that are powers of 2 (i.e. 1, 2, 4, 8, …, 128), and union labels are created by ORing base labels.

The following program demonstrates label propagation by checking that the correct labels are propagated.

```
#include <sanitizer/dfsan_interface.h>
#include <assert.h>

int main(void) {
  int i = 100;
  int j = 200;
  int k = 300;
  dfsan_label i_label = 1;
```

```
  dfsan_label j_label = 2;
  dfsan_label k_label = 4;
  dfsan_set_label(i_label, &i, sizeof(i));
  dfsan_set_label(j_label, &j, sizeof(j));
  dfsan_set_label(k_label, &k, sizeof(k));

  dfsan_label ij_label = dfsan_get_label(i + j);

  assert(ij_label & i_label);  // ij_label has i_label
  assert(ij_label & j_label);  // ij_label has j_label
  assert(!(ij_label & k_label));  // ij_label doesn't have k_label
  assert(ij_label == 3);  // Verifies all of the above

  // Or, equivalently:
  assert(dfsan_has_label(ij_label, i_label));
  assert(dfsan_has_label(ij_label, j_label));
  assert(!dfsan_has_label(ij_label, k_label));

  dfsan_label ijk_label = dfsan_get_label(i + j + k);

  assert(ijk_label & i_label);  // ijk_label has i_label
  assert(ijk_label & j_label);  // ijk_label has j_label
  assert(ijk_label & k_label);  // ijk_label has k_label
  assert(ijk_label == 7);  // Verifies all of the above

  // Or, equivalently:
  assert(dfsan_has_label(ijk_label, i_label));
  assert(dfsan_has_label(ijk_label, j_label));
  assert(dfsan_has_label(ijk_label, k_label));

  return 0;
}
```

Origin Tracking

DataFlowSanitizer can track origins of labeled values. This feature is enabled by `-mllvm -dfsan-track-origins=1`. For example,

```
% cat test.cc
#include <sanitizer/dfsan_interface.h>
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 0;
  dfsan_set_label(i_label, &i, sizeof(i));
  int j = i + 1;
  dfsan_print_origin_trace(&j, "A flow from i to j");
  return 0;
}

% clang++ -fsanitize=dataflow -mllvm -dfsan-track-origins=1 -fno-omit-frame-pointer -g -O2 test.cc
% ./a.out
Taint value 0x1 (at 0x7ffd42bf415c) origin tracking (A flow from i to j)
Origin value: 0x13900001, Taint value was stored to memory at
  #0 0x55676db85a62 in main test.cc:7:7
  #1 0x7f0083611bbc in __libc_start_main libc-start.c:285

Origin value: 0x9e00001, Taint value was created at
  #0 0x55676db85a08 in main test.cc:6:3
  #1 0x7f0083611bbc in __libc_start_main libc-start.c:285
```

By `-mllvm -dfsan-track-origins=1` DataFlowSanitizer collects only intermediate stores a labeled value went through. Origin tracking slows down program execution by a factor of 2x on top of the usual DataFlowSanitizer

slowdown and increases memory overhead by 1x. By `-mllvm -dfsan-track-origins=2` DataFlowSanitizer also collects intermediate loads a labeled value went through. This mode slows down program execution by a factor of 4x.

Current status

DataFlowSanitizer is a work in progress, currently under development for x86_64 Linux.

Design

Please refer to the design document.

LeakSanitizer

Introduction 663

Usage 663

Supported Platforms 663

More Information 664

Introduction

LeakSanitizer is a run-time memory leak detector. It can be combined with AddressSanitizer to get both memory error and leak detection, or used in a stand-alone mode. LSan adds almost no performance overhead until the very end of the process, at which point there is an extra leak detection phase.

Usage

AddressSanitizer: integrates LeakSanitizer and enables it by default on supported platforms.

```
$ cat memory-leak.c
#include <stdlib.h>
void *p;
int main() {
  p = malloc(7);
  p = 0; // The memory is leaked here.
  return 0;
}
% clang -fsanitize=address -g memory-leak.c ; ASAN_OPTIONS=detect_leaks=1 ./a.out
==23646==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 7 byte(s) in 1 object(s) allocated from:
    #0 0x4af01b in __interceptor_malloc /projects/compiler-rt/lib/asan/asan_malloc_linux.cc:52:3
    #1 0x4da26a in main memory-leak.c:4:7
    #2 0x7f076fd9cec4 in __libc_start_main libc-start.c:287
SUMMARY: AddressSanitizer: 7 byte(s) leaked in 1 allocation(s).
```

To use LeakSanitizer in stand-alone mode, link your program with `-fsanitize=leak` flag. Make sure to use `clang` (not `ld`) for the link step, so that it would link in proper LeakSanitizer run-time library into the final executable.

Supported Platforms

- Android aarch64/i386/x86_64
- Fuchsia aarch64/x86_64
- Linux arm/aarch64/mips64/ppc64/ppc64le/riscv64/s390x/i386/x86_64
- macOS aarch64/i386/x86_64
- NetBSD i386/x86_64

More Information

https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer

SanitizerCoverage

Introduction 664
Tracing PCs with guards 664
Inline 8bit-counters 666
Inline bool-flag 666
PC-Table 666
Tracing PCs 667
Instrumentation points 667
Edge coverage 667
Tracing data flow 667
Disabling instrumentation with `__attribute__((no_sanitize("coverage")))` 668
Disabling instrumentation without source modification 669
Default implementation 669
Sancov data format 670
Sancov Tool 670
Coverage Reports 670
Output directory 670

Introduction

LLVM has a simple code coverage instrumentation built in (SanitizerCoverage). It inserts calls to user-defined functions on function-, basic-block-, and edge- levels. Default implementations of those callbacks are provided and implement simple coverage reporting and visualization, however if you need *just* coverage visualization you may want to use SourceBasedCodeCoverage instead.

Tracing PCs with guards

With `-fsanitize-coverage=trace-pc-guard` the compiler will insert the following code on every edge:

```
__sanitizer_cov_trace_pc_guard(&guard_variable)
```

Every edge will have its own *guard_variable* (uint32_t).

The compiler will also insert calls to a module constructor:

```
// The guards are [start, stop).
// This function will be called at least once per DSO and may be called
// more than once with the same values of start/stop.
__sanitizer_cov_trace_pc_guard_init(uint32_t *start, uint32_t *stop);
```

With an additional `...=trace-pc,indirect-calls` flag `__sanitizer_cov_trace_pc_indirect(void *callee)` will be inserted on every indirect call.

The functions *__sanitizer_cov_trace_pc_** should be defined by the user.

Example:

```
// trace-pc-guard-cb.cc
#include <stdint.h>
#include <stdio.h>
#include <sanitizer/coverage_interface.h>

// This callback is inserted by the compiler as a module constructor
```

```
// into every DSO. 'start' and 'stop' correspond to the
// beginning and end of the section with the guards for the entire
// binary (executable or DSO). The callback will be called at least
// once per DSO and may be called multiple times with the same parameters.
extern "C" void __sanitizer_cov_trace_pc_guard_init(uint32_t *start,
                                                    uint32_t *stop) {
  static uint64_t N;  // Counter for the guards.
  if (start == stop || *start) return;  // Initialize only once.
  printf("INIT: %p %p\n", start, stop);
  for (uint32_t *x = start; x < stop; x++)
    *x = ++N;  // Guards should start from 1.
}

// This callback is inserted by the compiler on every edge in the
// control flow (some optimizations apply).
// Typically, the compiler will emit the code like this:
//    if(*guard)
//      __sanitizer_cov_trace_pc_guard(guard);
// But for large functions it will emit a simple call:
//    __sanitizer_cov_trace_pc_guard(guard);
extern "C" void __sanitizer_cov_trace_pc_guard(uint32_t *guard) {
  if (!*guard) return;  // Duplicate the guard check.
  // If you set *guard to 0 this code will not be called again for this edge.
  // Now you can get the PC and do whatever you want:
  //   store it somewhere or symbolize it and print right away.
  // The values of `*guard` are as you set them in
  // __sanitizer_cov_trace_pc_guard_init and so you can make them consecutive
  // and use them to dereference an array or a bit vector.
  void *PC = __builtin_return_address(0);
  char PcDescr[1024];
  // This function is a part of the sanitizer run-time.
  // To use it, link with AddressSanitizer or other sanitizer.
  __sanitizer_symbolize_pc(PC, "%p %F %L", PcDescr, sizeof(PcDescr));
  printf("guard: %p %x PC %s\n", guard, *guard, PcDescr);
}

// trace-pc-guard-example.cc
void foo() { }
int main(int argc, char **argv) {
  if (argc > 1) foo();
}
```

```
clang++ -g  -fsanitize-coverage=trace-pc-guard trace-pc-guard-example.cc -c
clang++ trace-pc-guard-cb.cc trace-pc-guard-example.o -fsanitize=address
ASAN_OPTIONS=strip_path_prefix=`pwd`/ ./a.out


INIT: 0x71bcd0 0x71bce0
guard: 0x71bcd4 2 PC 0x4ecd5b in main trace-pc-guard-example.cc:2
guard: 0x71bcd8 3 PC 0x4ecd9e in main trace-pc-guard-example.cc:3:7


ASAN_OPTIONS=strip_path_prefix=`pwd`/ ./a.out with-foo


INIT: 0x71bcd0 0x71bce0
guard: 0x71bcd4 2 PC 0x4ecd5b in main trace-pc-guard-example.cc:3
guard: 0x71bcdc 4 PC 0x4ecdc7 in main trace-pc-guard-example.cc:4:17
```

```
guard: 0x71bcd0 1 PC 0x4ecd20 in foo() trace-pc-guard-example.cc:2:14
```

Inline 8bit-counters

Experimental, may change or disappear in future

With `-fsanitize-coverage=inline-8bit-counters` the compiler will insert inline counter increments on every edge. This is similar to `-fsanitize-coverage=trace-pc-guard` but instead of a callback the instrumentation simply increments a counter.

Users need to implement a single function to capture the counters at startup.

```
extern "C"
void __sanitizer_cov_8bit_counters_init(char *start, char *end) {
  // [start,end) is the array of 8-bit counters created for the current DSO.
  // Capture this array in order to read/modify the counters.
}
```

Inline bool-flag

Experimental, may change or disappear in future

With `-fsanitize-coverage=inline-bool-flag` the compiler will insert setting an inline boolean to true on every edge. This is similar to `-fsanitize-coverage=inline-8bit-counter` but instead of an increment of a counter, it just sets a boolean to true.

Users need to implement a single function to capture the flags at startup.

```
extern "C"
void __sanitizer_cov_bool_flag_init(bool *start, bool *end) {
  // [start,end) is the array of boolean flags created for the current DSO.
  // Capture this array in order to read/modify the flags.
}
```

PC-Table

Experimental, may change or disappear in future

Note: this instrumentation might be incompatible with dead code stripping (`-Wl,-gc-sections`) for linkers other than LLD, thus resulting in a significant binary size overhead. For more information, see Bug 34636.

With `-fsanitize-coverage=pc-table` the compiler will create a table of instrumented PCs. Requires either `-fsanitize-coverage=inline-8bit-counters`, or `-fsanitize-coverage=inline-bool-flag`, or `-fsanitize-coverage=trace-pc-guard`.

Users need to implement a single function to capture the PC table at startup:

```
extern "C"
void __sanitizer_cov_pcs_init(const uintptr_t *pcs_beg,
                              const uintptr_t *pcs_end) {
  // [pcs_beg,pcs_end) is the array of ptr-sized integers representing
  // pairs [PC,PCFlags] for every instrumented block in the current DSO.
  // Capture this array in order to read the PCs and their Flags.
  // The number of PCs and PCFlags for a given DSO is the same as the number
  // of 8-bit counters (-fsanitize-coverage=inline-8bit-counters), or
  // boolean flags (-fsanitize-coverage=inline=bool-flags), or trace_pc_guard
  // callbacks (-fsanitize-coverage=trace-pc-guard).
  // A PCFlags describes the basic block:
  //  * bit0: 1 if the block is the function entry block, 0 otherwise.
}
```

Tracing PCs

With `-fsanitize-coverage=trace-pc` the compiler will insert `__sanitizer_cov_trace_pc()` on every edge. With an additional `...=trace-pc,indirect-calls` flag `__sanitizer_cov_trace_pc_indirect(void *callee)` will be inserted on every indirect call. These callbacks are not implemented in the Sanitizer run-time and should be defined by the user. This mechanism is used for fuzzing the Linux kernel (https://github.com/google/syzkaller).

Instrumentation points

Sanitizer Coverage offers different levels of instrumentation.

- `edge` (default): edges are instrumented (see below).
- `bb`: basic blocks are instrumented.
- `func`: only the entry block of every function will be instrumented.

Use these flags together with `trace-pc-guard` or `trace-pc`, like this: `-fsanitize-coverage=func,trace-pc-guard`.

When `edge` or `bb` is used, some of the edges/blocks may still be left uninstrumented (pruned) if such instrumentation is considered redundant. Use `no-prune` (e.g. `-fsanitize-coverage=bb,no-prune,trace-pc-guard`) to disable pruning. This could be useful for better coverage visualization.

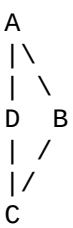
Edge coverage

Consider this code:

```
void foo(int *a) {
  if (a)
    *a = 0;
}
```

It contains 3 basic blocks, let's name them A, B, C:

If blocks A, B, and C are all covered we know for certain that the edges A=>B and B=>C were executed, but we still don't know if the edge A=>C was executed. Such edges of control flow graph are called critical. The edge-level coverage simply splits all critical edges by introducing new dummy blocks and then instruments those blocks:



Tracing data flow

Support for data-flow-guided fuzzing. With `-fsanitize-coverage=trace-cmp` the compiler will insert extra instrumentation around comparison instructions and switch statements. Similarly, with `-fsanitize-coverage=trace-div` the compiler will instrument integer division instructions (to capture the right argument of division) and with `-fsanitize-coverage=trace-gep` – the LLVM GEP instructions (to capture array indices). Similarly, with `-fsanitize-coverage=trace-loads` and `-fsanitize-coverage=trace-stores` the compiler will instrument loads and stores, respectively.

Currently, these flags do not work by themselves - they require one of `-fsanitize-coverage={trace-pc,inline-8bit-counters,inline-bool}` flags to work.

Unless `no-prune` option is provided, some of the comparison instructions will not be instrumented.

```
// Called before a comparison instruction.
// Arg1 and Arg2 are arguments of the comparison.
void __sanitizer_cov_trace_cmp1(uint8_t Arg1, uint8_t Arg2);
void __sanitizer_cov_trace_cmp2(uint16_t Arg1, uint16_t Arg2);
void __sanitizer_cov_trace_cmp4(uint32_t Arg1, uint32_t Arg2);
void __sanitizer_cov_trace_cmp8(uint64_t Arg1, uint64_t Arg2);

// Called before a comparison instruction if exactly one of the arguments is constant.
// Arg1 and Arg2 are arguments of the comparison, Arg1 is a compile-time constant.
// These callbacks are emitted by -fsanitize-coverage=trace-cmp since 2017-08-11
void __sanitizer_cov_trace_const_cmp1(uint8_t Arg1, uint8_t Arg2);
void __sanitizer_cov_trace_const_cmp2(uint16_t Arg1, uint16_t Arg2);
void __sanitizer_cov_trace_const_cmp4(uint32_t Arg1, uint32_t Arg2);
void __sanitizer_cov_trace_const_cmp8(uint64_t Arg1, uint64_t Arg2);

// Called before a switch statement.
// Val is the switch operand.
// Cases[0] is the number of case constants.
// Cases[1] is the size of Val in bits.
// Cases[2:] are the case constants.
void __sanitizer_cov_trace_switch(uint64_t Val, uint64_t *Cases);

// Called before a division statement.
// Val is the second argument of division.
void __sanitizer_cov_trace_div4(uint32_t Val);
void __sanitizer_cov_trace_div8(uint64_t Val);

// Called before a GetElememementPtr (GEP) instruction
// for every non-constant array index.
void __sanitizer_cov_trace_gep(uintptr_t Idx);

// Called before a load of appropriate size. Addr is the address of the load.
void __sanitizer_cov_load1(uint8_t *addr);
void __sanitizer_cov_load2(uint16_t *addr);
void __sanitizer_cov_load4(uint32_t *addr);
void __sanitizer_cov_load8(uint64_t *addr);
void __sanitizer_cov_load16(__int128 *addr);
// Called before a store of appropriate size. Addr is the address of the store.
void __sanitizer_cov_store1(uint8_t *addr);
void __sanitizer_cov_store2(uint16_t *addr);
void __sanitizer_cov_store4(uint32_t *addr);
void __sanitizer_cov_store8(uint64_t *addr);
void __sanitizer_cov_store16(__int128 *addr);
```

Disabling instrumentation with `__attribute__((no_sanitize("coverage")))`

It is possible to disable coverage instrumentation for select functions via the function attribute `__attribute__((no_sanitize("coverage")))`. Because this attribute may not be supported by other compilers, it is recommended to use it together with `__has_feature(coverage_sanitizer)`.

Disabling instrumentation without source modification

It is sometimes useful to tell SanitizerCoverage to instrument only a subset of the functions in your target without modifying source files. With `-fsanitize-coverage-allowlist=allowlist.txt` and `-fsanitize-coverage-ignorelist=blocklist.txt`, you can specify such a subset through the combination of an allowlist and a blocklist.

SanitizerCoverage will only instrument functions that satisfy two conditions. First, the function should belong to a source file with a path that is both allowlisted and not blocklisted. Second, the function should have a mangled name that is both allowlisted and not blocklisted.

The allowlist and blocklist format is similar to that of the sanitizer blocklist format. The default allowlist will match every source file and every function. The default blocklist will match no source file and no function.

A common use case is to have the allowlist list folders or source files for which you want instrumentation and allow all function names, while the blocklist will opt out some specific files or functions that the allowlist loosely allowed.

Here is an example allowlist:

```
# Enable instrumentation for a whole folder
src:bar/*
# Enable instrumentation for a specific source file
src:foo/a.cpp
# Enable instrumentation for all functions in those files
fun:*
```

And an example blocklist:

```
# Disable instrumentation for a specific source file that the allowlist allowed
src:bar/b.cpp
# Disable instrumentation for a specific function that the allowlist allowed
fun:*myFunc*
```

The use of * wildcards above is required because function names are matched after mangling. Without the wildcards, one would have to write the whole mangled name.

Be careful that the paths of source files are matched exactly as they are provided on the clang command line. For example, the allowlist above would include file `bar/b.cpp` if the path was provided exactly like this, but would it would fail to include it with other ways to refer to the same file such as `./bar/b.cpp`, or `bar\b.cpp` on Windows. So, please make sure to always double check that your lists are correctly applied.

Default implementation

The sanitizer run-time (AddressSanitizer, MemorySanitizer, etc) provide a default implementations of some of the coverage callbacks. You may use this implementation to dump the coverage on disk at the process exit.

Example:

```
% cat -n cov.cc
     1  #include <stdio.h>
     2  __attribute__((noinline))
     3  void foo() { printf("foo\n"); }
     4
     5  int main(int argc, char **argv) {
     6    if (argc == 2)
     7      foo();
     8    printf("main\n");
     9  }
% clang++ -g cov.cc -fsanitize=address -fsanitize-coverage=trace-pc-guard
% ASAN_OPTIONS=coverage=1 ./a.out; wc -c *.sancov
main
SanitizerCoverage: ./a.out.7312.sancov 2 PCs written
24 a.out.7312.sancov
% ASAN_OPTIONS=coverage=1 ./a.out foo ; wc -c *.sancov
foo
```

```
main
SanitizerCoverage: ./a.out.7316.sancov 3 PCs written
24 a.out.7312.sancov
32 a.out.7316.sancov
```

Every time you run an executable instrumented with SanitizerCoverage one `*.sancov` file is created during the process shutdown. If the executable is dynamically linked against instrumented DSOs, one `*.sancov` file will be also created for every DSO.

Sancov data format

The format of `*.sancov` files is very simple: the first 8 bytes is the magic, one of `0xC0BFFFFFFFFFFF64` and `0xC0BFFFFFFFFFFF32`. The last byte of the magic defines the size of the following offsets. The rest of the data is the offsets in the corresponding binary/DSO that were executed during the run.

Sancov Tool

An simple `sancov` tool is provided to process coverage files. The tool is part of LLVM project and is currently supported only on Linux. It can handle symbolization tasks autonomously without any extra support from the environment. You need to pass .sancov files (named `<module_name>.<pid>.sancov` and paths to all corresponding binary elf files. Sancov matches these files using module names and binaries file names.

```
USAGE: sancov [options] <action> (<binary file>|<.sancov file>)...

Action (required)
  -print                    - Print coverage addresses
  -covered-functions        - Print all covered functions.
  -not-covered-functions    - Print all not covered functions.
  -symbolize                - Symbolizes the report.

Options
  -blocklist=<string>         - Blocklist file (sanitizer blocklist format).
  -demangle                   - Print demangled function name.
  -strip_path_prefix=<string> - Strip this prefix from file paths in reports
```

Coverage Reports

Experimental

`.sancov` files do not contain enough information to generate a source-level coverage report. The missing information is contained in debug info of the binary. Thus the `.sancov` has to be symbolized to produce a `.symcov` file first:

```
sancov -symbolize my_program.123.sancov my_program > my_program.123.symcov
```

The `.symcov` file can be browsed overlaid over the source code by running `tools/sancov/coverage-report-server.py` script that will start an HTTP server.

Output directory

By default, .sancov files are created in the current working directory. This can be changed with `ASAN_OPTIONS=coverage_dir=/path`:

```
% ASAN_OPTIONS="coverage=1:coverage_dir=/tmp/cov" ./a.out foo
% ls -l /tmp/cov/*sancov
-rw-r----- 1 kcc eng 4 Nov 27 12:21 a.out.22673.sancov
-rw-r----- 1 kcc eng 8 Nov 27 12:21 a.out.22679.sancov
```

SanitizerStats

Introduction 671
How to build and run 671

Introduction

The sanitizers support a simple mechanism for gathering profiling statistics to help understand the overhead associated with sanitizers.

How to build and run

SanitizerStats can currently only be used with Control Flow Integrity. In addition to `-fsanitize=cfi*`, pass the `-fsanitize-stats` flag. This will cause the program to count the number of times that each control flow integrity check in the program fires.

At run time, set the `SANITIZER_STATS_PATH` environment variable to direct statistics output to a file. The file will be written on process exit. The following substitutions will be applied to the environment variable:

- %b – The executable basename.
- %p – The process ID.

You can also send the `SIGUSR2` signal to a process to make it write sanitizer statistics immediately.

The `sanstats` program can be used to dump statistics. It takes as a command line argument the path to a statistics file produced by a program compiled with `-fsanitize-stats`.

The output of `sanstats` is in four columns, separated by spaces. The first column is the file and line number of the call site. The second column is the function name. The third column is the type of statistic gathered (in this case, the type of control flow integrity check). The fourth column is the call count.

Example:

```
$ cat -n vcall.cc
     1 struct A {
     2   virtual void f() {}
     3 };
     4
     5 __attribute__((noinline)) void g(A *a) {
     6   a->f();
     7 }
     8
     9 int main() {
    10   A a;
    11   g(&a);
    12 }
$ clang++ -fsanitize=cfi -fvisibility=hidden -flto -fuse-ld=gold vcall.cc -fsanitize-stats -g
$ SANITIZER_STATS_PATH=a.stats ./a.out
$ sanstats a.stats
vcall.cc:6 _Z1gP1A cfi-vcall 1
```

Sanitizer special case list

Introduction 672
Goal and usage 672
Example 672
Format 672

Introduction

This document describes the way to disable or alter the behavior of sanitizer tools for certain source-level entities by providing a special file at compile-time.

Goal and usage

User of sanitizer tools, such as AddressSanitizer, ThreadSanitizer or MemorySanitizer may want to disable or alter some checks for certain source-level entities to:

- speedup hot function, which is known to be correct;
- ignore a function that does some low-level magic (e.g. walks through the thread stack, bypassing the frame boundaries);
- ignore a known problem.

To achieve this, user may create a file listing the entities they want to ignore, and pass it to clang at compile-time using `-fsanitize-ignorelist` flag. See Clang Compiler User's Manual for details.

Example

```
$ cat foo.c
#include <stdlib.h>
void bad_foo() {
  int *a = (int*)malloc(40);
  a[10] = 1;
}
int main() { bad_foo(); }
$ cat ignorelist.txt
# Ignore reports from bad_foo function.
fun:bad_foo
$ clang -fsanitize=address foo.c ; ./a.out
# AddressSanitizer prints an error report.
$ clang -fsanitize=address -fsanitize-ignorelist=ignorelist.txt foo.c ; ./a.out
# No error report here.
```

Format

Ignorelists consist of entries, optionally grouped into sections. Empty lines and lines starting with "#" are ignored.

Section names are regular expressions written in square brackets that denote which sanitizer the following entries apply to. For example, `[address]` specifies AddressSanitizer while `[cfi-vcall|cfi-icall]` specifies Control Flow Integrity virtual and indirect call checking. Entries without a section will be placed under the `[*]` section applying to all enabled sanitizers.

Entries contain an entity type, followed by a colon and a regular expression, specifying the names of the entities, optionally followed by an equals sign and a tool-specific category, e.g. `fun:*ExampleFunc=example_category`. The meaning of * in regular expression for entity names is different - it is treated as in shell wildcarding. Two generic entity types are `src` and `fun`, which allow users to specify source files and functions, respectively. Some sanitizer tools may introduce custom entity types and categories - refer to tool-specific docs.

```
# Lines starting with # are ignored.
# Turn off checks for the source file (use absolute path or path relative
# to the current working directory):
src:/path/to/source/file.c
# Turn off checks for a particular functions (use mangled names):
fun:MyFooBar
fun:_Z8MyFooBarv
# Extended regular expressions are supported:
fun:bad_(foo|bar)
src:bad_source[1-9].c
# Shell like usage of * is supported (* is treated as .*):
```

```
src:bad/sources/*
fun:*BadFunction*
# Specific sanitizer tools may introduce categories.
src:/special/path/*=special_sources
# Sections can be used to limit ignorelist entries to specific sanitizers
[address]
fun:*BadASanFunc*
# Section names are regular expressions
[cfi-vcall|cfi-icall]
fun:*BadCfiCall
# Entries without sections are placed into [*] and apply to all sanitizers
```

Control Flow Integrity

Control Flow Integrity Design Documentation

This page documents the design of the Control Flow Integrity schemes supported by Clang.

Forward-Edge CFI for Virtual Calls

This scheme works by allocating, for each static type used to make a virtual call, a region of read-only storage in the object file holding a bit vector that maps onto to the region of storage used for those virtual tables. Each set bit in the bit vector corresponds to the address point for a virtual table compatible with the static type for which the bit vector is being built.

For example, consider the following three C++ classes:

```
struct A {
  virtual void f1();
  virtual void f2();
  virtual void f3();
};

struct B : A {
  virtual void f1();
  virtual void f2();
  virtual void f3();
};

struct C : A {
  virtual void f1();
  virtual void f2();
  virtual void f3();
};
```

The scheme will cause the virtual tables for A, B and C to be laid out consecutively:

Virtual Table Layout for A, B, C

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A::offset-to-top	&A::rtti	&A::f1	&A::f2	&A::f3	B::offset-to-top	&B::rtti	&B::f1	&B::f2	&B::f3	C::offset-to-top	&C::rtti	&C::f1	&C::f2	&C::f3

The bit vector for static types A, B and C will look like this:

Bit Vectors for A, B, C

Class	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0
B	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
C	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

Bit vectors are represented in the object file as byte arrays. By loading from indexed offsets into the byte array and applying a mask, a program can test bits from the bit set with a relatively short instruction sequence. Bit vectors may overlap so long as they use different bits. For the full details, see the ByteArrayBuilder class.

In this case, assuming A is laid out at offset 0 in bit 0, B at offset 0 in bit 1 and C at offset 0 in bit 2, the byte array would look like this:

```
char bits[] = { 0, 0, 1, 0, 0, 0, 3, 0, 0, 0, 0, 5, 0, 0 };
```

To emit a virtual call, the compiler will assemble code that checks that the object's virtual table pointer is in-bounds and aligned and that the relevant bit is set in the bit vector.

For example on x86 a typical virtual call may look like this:

```
ca7fbb:       48 8b 0f                mov    (%rdi),%rcx
ca7fbe:       48 8d 15 c3 42 fb 07    lea    0x7fb42c3(%rip),%rdx
ca7fc5:       48 89 c8                mov    %rcx,%rax
ca7fc8:       48 29 d0                sub    %rdx,%rax
ca7fcb:       48 c1 c0 3d             rol    $0x3d,%rax
ca7fcf:       48 3d 7f 01 00 00       cmp    $0x17f,%rax
ca7fd5:       0f 87 36 05 00 00       ja     ca8511
ca7fdb:       48 8d 15 c0 0b f7 06    lea    0x6f70bc0(%rip),%rdx
ca7fe2:       f6 04 10 10             testb  $0x10,(%rax,%rdx,1)
ca7fe6:       0f 84 25 05 00 00       je     ca8511
ca7fec:       ff 91 98 00 00 00       callq  *0x98(%rcx)
  [...]
ca8511:       0f 0b                   ud2
```

The compiler relies on co-operation from the linker in order to assemble the bit vectors for the whole program. It currently does this using LLVM's type metadata mechanism together with link-time optimization.

Optimizations

The scheme as described above is the fully general variant of the scheme. Most of the time we are able to apply one or more of the following optimizations to improve binary size or performance.

In fact, if you try the above example with the current version of the compiler, you will probably find that it will not use the described virtual table layout or machine instructions. Some of the optimizations we are about to introduce cause the compiler to use a different layout or a different sequence of machine instructions.

Stripping Leading/Trailing Zeros in Bit Vectors

If a bit vector contains leading or trailing zeros, we can strip them from the vector. The compiler will emit code to check if the pointer is in range of the region covered by ones, and perform the bit vector check using a truncated version of the bit vector. For example, the bit vectors for our example class hierarchy will be emitted like this:

Bit Vectors for A, B, C

Class	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A			1	0	0	0	0	1	0	0	0	0	1		
B								1							
C													1		

Short Inline Bit Vectors

If the vector is sufficiently short, we can represent it as an inline constant on x86. This saves us a few instructions when reading the correct element of the bit vector.

If the bit vector fits in 32 bits, the code looks like this:

```
 dc2:       48 8b 03                mov    (%rbx),%rax
 dc5:       48 8d 15 14 1e 00 00    lea    0x1e14(%rip),%rdx
 dcc:       48 89 c1                mov    %rax,%rcx
 dcf:       48 29 d1                sub    %rdx,%rcx
 dd2:       48 c1 c1 3d             rol    $0x3d,%rcx
 dd6:       48 83 f9 03             cmp    $0x3,%rcx
 dda:       77 2f                   ja     e0b <main+0x9b>
 ddc:       ba 09 00 00 00          mov    $0x9,%edx
 de1:       0f a3 ca                bt     %ecx,%edx
 de4:       73 25                   jae    e0b <main+0x9b>
 de6:       48 89 df                mov    %rbx,%rdi
 de9:       ff 10                   callq  *(%rax)
[...]
 e0b:       0f 0b                   ud2
```

Or if the bit vector fits in 64 bits:

```
11a6:       48 8b 03                mov    (%rbx),%rax
11a9:       48 8d 15 d0 28 00 00    lea    0x28d0(%rip),%rdx
11b0:       48 89 c1                mov    %rax,%rcx
11b3:       48 29 d1                sub    %rdx,%rcx
11b6:       48 c1 c1 3d             rol    $0x3d,%rcx
11ba:       48 83 f9 2a             cmp    $0x2a,%rcx
11be:       77 35                   ja     11f5 <main+0xb5>
11c0:       48 ba 09 00 00 00 00    movabs $0x40000000009,%rdx
11c7:       04 00 00
11ca:       48 0f a3 ca             bt     %rcx,%rdx
11ce:       73 25                   jae    11f5 <main+0xb5>
11d0:       48 89 df                mov    %rbx,%rdi
11d3:       ff 10                   callq  *(%rax)
[...]
11f5:       0f 0b                   ud2
```

If the bit vector consists of a single bit, there is only one possible virtual table, and the check can consist of a single equality comparison:

```
9a2:   48 8b 03                mov    (%rbx),%rax
9a5:   48 8d 0d a4 13 00 00    lea    0x13a4(%rip),%rcx
9ac:   48 39 c8                cmp    %rcx,%rax
9af:   75 25                   jne    9d6 <main+0x86>
9b1:   48 89 df                mov    %rbx,%rdi
9b4:   ff 10                   callq  *(%rax)
[...]
9d6:   0f 0b                   ud2
```

Virtual Table Layout

The compiler lays out classes of disjoint hierarchies in separate regions of the object file. At worst, bit vectors in disjoint hierarchies only need to cover their disjoint hierarchy. But the closer that classes in sub-hierarchies are laid out to each other, the smaller the bit vectors for those sub-hierarchies need to be (see “Stripping Leading/Trailing Zeros in Bit Vectors” above). The GlobalLayoutBuilder class is responsible for laying out the globals efficiently to minimize the sizes of the underlying bitsets.

Alignment

If all gaps between address points in a particular bit vector are multiples of powers of 2, the compiler can compress the bit vector by strengthening the alignment requirements of the virtual table pointer. For example, given this class hierarchy:

```
struct A {
  virtual void f1();
  virtual void f2();
};

struct B : A {
  virtual void f1();
  virtual void f2();
  virtual void f3();
  virtual void f4();
  virtual void f5();
  virtual void f6();
};

struct C : A {
  virtual void f1();
  virtual void f2();
};
```

The virtual tables will be laid out like this:

Virtual Table Layout for A, B, C

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A::offset-to-top	&A::rtti	&A::f1	&A::f2	B::offset-to-top	&B::rtti	&B::f1	&B::f2	&B::f3	&B::f4	&B::f5	&B::f6	C::offset-to-top	&C::rtti	&C::f1	&C::f2

Notice that each address point for A is separated by 4 words. This lets us emit a compressed bit vector for A that looks like this:

2	6	10	14
1	1	0	1

At call sites, the compiler will strengthen the alignment requirements by using a different rotate count. For example, on a 64-bit machine where the address points are 4-word aligned (as in A from our example), the `rol` instruction may look like this:

```
dd2:       48 c1 c1 3b             rol    $0x3b,%rcx
```

Padding to Powers of 2

Of course, this alignment scheme works best if the address points are in fact aligned correctly. To make this more likely to happen, we insert padding between virtual tables that in many cases aligns address points to a power of 2. Specifically, our padding aligns virtual tables to the next highest power of 2 bytes; because address points for specific base classes normally appear at fixed offsets within the virtual table, this normally has the effect of aligning the address points as well.

This scheme introduces tradeoffs between decreased space overhead for instructions and bit vectors and increased overhead in the form of padding. We therefore limit the amount of padding so that we align to no more than 128 bytes. This number was found experimentally to provide a good tradeoff.

Eliminating Bit Vector Checks for All-Ones Bit Vectors

If the bit vector is all ones, the bit vector check is redundant; we simply need to check that the address is in range and well aligned. This is more likely to occur if the virtual tables are padded.

Forward-Edge CFI for Virtual Calls by Interleaving Virtual Tables

Dimitar et. al. proposed a novel approach that interleaves virtual tables in [4]. This approach is more efficient in terms of space because padding and bit vectors are no longer needed. At the same time, it is also more efficient in terms of performance because in the interleaved layout address points of the virtual tables are consecutive, thus the validity check of a virtual vtable pointer is always a range check.

At a high level, the interleaving scheme consists of three steps: 1) split virtual table groups into separate virtual tables, 2) order virtual tables by a pre-order traversal of the class hierarchy and 3) interleave virtual tables.

The interleaving scheme implemented in LLVM is inspired by [4] but has its own enhancements (more in Interleave virtual tables).

4(1, 2, 3) Protecting C++ Dynamic Dispatch Through VTable Interleaving. Dimitar Bounov, Rami Gökhan K■c■, Sorin Lerner.

Split virtual table groups into separate virtual tables

The Itanium C++ ABI glues multiple individual virtual tables for a class into a combined virtual table (virtual table group). The interleaving scheme, however, can only work with individual virtual tables so it must split the combined virtual tables first. In comparison, the old scheme does not require the splitting but it is more efficient when the combined virtual tables have been split. The GlobalSplit pass is responsible for splitting combined virtual tables into individual ones.

Order virtual tables by a pre-order traversal of the class hierarchy

This step is common to both the old scheme described above and the interleaving scheme. For the interleaving scheme, since the combined virtual tables have been split in the previous step, this step ensures that for any class all the compatible virtual tables will appear consecutively. For the old scheme, the same property may not hold since it may work on combined virtual tables.

For example, consider the following four C++ classes:

```
struct A {
  virtual void f1();
};

struct B : A {
  virtual void f1();
  virtual void f2();
};

struct C : A {
  virtual void f1();
  virtual void f3();
};

struct D : B {
  virtual void f1();
  virtual void f2();
};
```

This step will arrange the virtual tables for A, B, C, and D in the order of *vtable-of-A, vtable-of-B, vtable-of-D, vtable-of-C*.

Interleave virtual tables

This step is where the interleaving scheme deviates from the old scheme. Instead of laying out whole virtual tables in the previously computed order, the interleaving scheme lays out table entries of the virtual tables strategically to ensure the following properties:

1. offset-to-top and RTTI fields layout property

The Itanium C++ ABI specifies that offset-to-top and RTTI fields appear at the offsets behind the address point. Note that libraries like libcxxabi do assume this property.

2. virtual function entry layout property

For each virtual function the distance between an virtual table entry for this function and the corresponding address point is always the same. This property ensures that dynamic dispatch still works with the interleaving layout.

Note that the interleaving scheme in the CFI implementation guarantees both properties above whereas the original scheme proposed in [4] only guarantees the second property.

To illustrate how the interleaving algorithm works, let us continue with the running example. The algorithm first separates all the virtual table entries into two work lists. To do so, it starts by allocating two work lists, one initialized with all the offset-to-top entries of virtual tables in the order computed in the last step, one initialized with all the RTTI entries in the same order.

Work list 1 Layout

0	1	2	3
A::offset-to-top	B::offset-to-top	D::offset-to-top	C::offset-to-top

Work list 2 layout

0	1	2	3	
&A::rtti	&B::rtti	&D::rtti	&C::rtti	

Then for each virtual function the algorithm goes through all the virtual tables in the previously computed order to collect all the related entries into a virtual function list. After this step, there are the following virtual function lists:

f1 list

0	1	2	3
&A::f1	&B::f1	&D::f1	&C::f1

f2 list

0	1
&B::f2	&D::f2

f3 list

0
&C::f3

Next, the algorithm picks the longest remaining virtual function list and appends the whole list to the shortest work list until no function lists are left, and pads the shorter work list so that they are of the same length. In the example, f1 list will be first added to work list 1, then f2 list will be added to work list 2, and finally f3 list will be added to the work list 2. Since work list 1 now has one more entry than work list 2, a padding entry is added to the latter. After this step, the two work lists look like:

Work list 1 Layout

0	1	2	3	4	5	6	7
A::offset-to-top	B::offset-to-top	D::offset-to-top	C::offset-to-top	&A::f1	&B::f1	&D::f1	&C::f1

Work list 2 layout

0	1	2	3	4	5	6	7
&A::rtti	&B::rtti	&D::rtti	&C::rtti	&B::f2	&D::f2	&C::f3	padding

Finally, the algorithm merges the two work lists into the interleaved layout by alternatingly moving the head of each list to the final layout. After this step, the final interleaved layout looks like:

Interleaved layout

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A::offset-to-top	&A::rtti	B::offset-to-top	&B::rtti	D::offset-to-top	&D::rtti	C::offset-to-top	&C::rtti	&A::f1	&B::f2	&B::f1	&D::f2	&D::f1	&C::f3	&C::f1	padding

In the above interleaved layout, each virtual table's offset-to-top and RTTI are always adjacent, which shows that the layout has the first property. For the second property, let us look at f2 as an example. In the interleaved layout, there are two entries for f2: B::f2 and D::f2. The distance between &B::f2 and its address point D::offset-to-top (the entry immediately after &B::rtti) is 5 entry-length, so is the distance between &D::f2 and C::offset-to-top (the entry immediately after &D::rtti).

Forward-Edge CFI for Indirect Function Calls

Under forward-edge CFI for indirect function calls, each unique function type has its own bit vector, and at each call site we need to check that the function pointer is a member of the function type's bit vector. This scheme works in a similar way to forward-edge CFI for virtual calls, the distinction being that we need to build bit vectors of function entry points rather than of virtual tables.

Unlike when re-arranging global variables, we cannot re-arrange functions in a particular order and base our calculations on the layout of the functions' entry points, as we have no idea how large a particular function will end up being (the function sizes could even depend on how we arrange the functions). Instead, we build a jump table, which is a block of code consisting of one branch instruction for each of the functions in the bit set that branches to the target function, and redirect any taken function addresses to the corresponding jump table entry. In this way, the distance between function entry points is predictable and controllable. In the object file's symbol table, the symbols for the target functions also refer to the jump table entries, so that addresses taken outside the module will pass any verification done inside the module.

In more concrete terms, suppose we have three functions `f`, g, h which are all of the same type, and a function foo that returns their addresses:

```
f:
mov 0, %eax
ret

g:
mov 1, %eax
ret

h:
mov 2, %eax
ret

foo:
```

```
mov f, %eax
mov g, %edx
mov h, %ecx
ret
```

Our jump table will (conceptually) look like this:

```
f:
jmp .Ltmp0 ; 5 bytes
int3       ; 1 byte
int3       ; 1 byte
int3       ; 1 byte

g:
jmp .Ltmp1 ; 5 bytes
int3       ; 1 byte
int3       ; 1 byte
int3       ; 1 byte

h:
jmp .Ltmp2 ; 5 bytes
int3       ; 1 byte
int3       ; 1 byte
int3       ; 1 byte

.Ltmp0:
mov 0, %eax
ret

.Ltmp1:
mov 1, %eax
ret

.Ltmp2:
mov 2, %eax
ret

foo:
mov f, %eax
mov g, %edx
mov h, %ecx
ret
```

Because the addresses of `f`, `g`, `h` are evenly spaced at a power of 2, and function types do not overlap (unlike class types with base classes), we can normally apply the Alignment and Eliminating Bit Vector Checks for All-Ones Bit Vectors optimizations thus simplifying the check at each call site to a range and alignment check.

Shared library support

EXPERIMENTAL

The basic CFI mode described above assumes that the application is a monolithic binary; at least that all possible virtual/indirect call targets and the entire class hierarchy are known at link time. The cross-DSO mode, enabled with **-f[no-]sanitize-cfi-cross-dso** relaxes this requirement by allowing virtual and indirect calls to cross the DSO boundary.

Assuming the following setup: the binary consists of several instrumented and several uninstrumented DSOs. Some of them may be dlopen-ed/dlclose-d periodically, even frequently.

- Calls made from uninstrumented DSOs are not checked and just work.
- Calls inside any instrumented DSO are fully protected.

- **Calls between different instrumented DSOs are also protected, with** a performance penalty (in addition to the monolithic CFI overhead).
- **Calls from an instrumented DSO to an uninstrumented one are** unchecked and just work, with performance penalty.
- **Calls from an instrumented DSO outside of any known DSO are** detected as CFI violations.

In the monolithic scheme a call site is instrumented as

```
if (!InlinedFastCheck(f))
  abort();
call *f
```

In the cross-DSO scheme it becomes

```
if (!InlinedFastCheck(f))
  __cfi_slowpath(CallSiteTypeId, f);
call *f
```

CallSiteTypeId

`CallSiteTypeId` is a stable process-wide identifier of the call-site type. For a virtual call site, the type in question is the class type; for an indirect function call it is the function signature. The mapping from a type to an identifier is an ABI detail. In the current, experimental, implementation the identifier of type T is calculated as follows:

- Obtain the mangled name for “typeinfo name for T”.
- Calculate MD5 hash of the name as a string.
- Reinterpret the first 8 bytes of the hash as a little-endian 64-bit integer.

It is possible, but unlikely, that collisions in the `CallSiteTypeId` hashing will result in weaker CFI checks that would still be conservatively correct.

CFI_Check

In the general case, only the target DSO knows whether the call to function `f` with type `CallSiteTypeId` is valid or not. To export this information, every DSO implements

```
void __cfi_check(uint64 CallSiteTypeId, void *TargetAddr, void *DiagData)
```

This function provides external modules with access to CFI checks for the targets inside this DSO. For each known `CallSiteTypeId`, this function performs an `llvm.type.test` with the corresponding type identifier. It reports an error if the type is unknown, or if the check fails. Depending on the values of compiler flags `-fsanitize-trap` and `-fsanitize-recover`, this function may print an error, abort and/or return to the caller. `DiagData` is an opaque pointer to the diagnostic information about the error, or `null` if the caller does not provide this information.

The basic implementation is a large switch statement over all values of CallSiteTypeId supported by this DSO, and each case is similar to the InlinedFastCheck() in the basic CFI mode.

CFI Shadow

To route CFI checks to the target DSO’s __cfi_check function, a mapping from possible virtual / indirect call targets to the corresponding __cfi_check functions is maintained. This mapping is implemented as a sparse array of 2 bytes for every possible page (4096 bytes) of memory. The table is kept readonly most of the time.

There are 3 types of shadow values:

- Address in a CFI-instrumented DSO.
- Unchecked address (a “trusted” non-instrumented DSO). Encoded as value 0xFFFF.
- Invalid address (everything else). Encoded as value 0.

For a CFI-instrumented DSO, a shadow value encodes the address of the __cfi_check function for all call targets in the corresponding memory page. If Addr is the target address, and V is the shadow value, then the address of __cfi_check is calculated as

```
__cfi_check = AlignUpTo(Addr, 4096) - (V + 1) * 4096
```

This works as long as __cfi_check is aligned by 4096 bytes and located below any call targets in its DSO, but not more than 256MB apart from them.

CFI_SlowPath

The slow path check is implemented in a runtime support library as

```
void __cfi_slowpath(uint64 CallSiteTypeId, void *TargetAddr)
void __cfi_slowpath_diag(uint64 CallSiteTypeId, void *TargetAddr, void *DiagData)
```

These functions loads a shadow value for `TargetAddr`, finds the address of `__cfi_check` as described above and calls that. `DiagData` is an opaque pointer to diagnostic data which is passed verbatim to `__cfi_check`, and `__cfi_slowpath` passes `nullptr` instead.

Compiler-RT library contains reference implementations of slowpath functions, but they have unresolvable issues with correctness and performance in the handling of dlopen(). It is recommended that platforms provide their own implementations, usually as part of libc or libdl.

Position-independent executable requirement

Cross-DSO CFI mode requires that the main executable is built as PIE. In non-PIE executables the address of an external function (taken from the main executable) is the address of that function's PLT record in the main executable. This would break the CFI checks.

Backward-edge CFI for return statements (RCFI)

This section is a proposal. As of March 2017 it is not implemented.

Backward-edge control flow (*RET* instructions) can be hijacked via overwriting the return address (*RA*) on stack. Various mitigation techniques (e.g. SafeStack, RFG, Intel CET) try to detect or prevent *RA* corruption on stack.

RCFI enforces the expected control flow in several different ways described below. RCFI heavily relies on LTO.

Leaf Functions

If *f()* is a leaf function (i.e. it has no calls except maybe no-return calls) it can be called using a special calling convention that stores *RA* in a dedicated register *R* before the *CALL* instruction. *f()* does not spill *R* and does not use the *RET* instruction, instead it uses the value in *R* to *JMP* to *RA*.

This flavour of CFI is *precise*, i.e. the function is guaranteed to return to the point exactly following the call.

An alternative approach is to copy *RA* from stack to *R* in the first instruction of *f()*, then *JMP* to *R*. This approach is simpler to implement (does not require changing the caller) but weaker (there is a small window when *RA* is actually stored on stack).

Functions called once

Suppose *f()* is called in just one place in the program (assuming we can verify this in LTO mode). In this case we can replace the *RET* instruction with a *JMP* instruction with the immediate constant for *RA*. This will *precisely* enforce the return control flow no matter what is stored on stack.

Another variant is to compare *RA* on stack with the known constant and abort if they don't match; then *JMP* to the known constant address.

Functions called in a small number of call sites

We may extend the above approach to cases where *f()* is called more than once (but still a small number of times). With LTO we know all possible values of *RA* and we check them one-by-one (or using binary search) against the value on stack. If the match is found, we *JMP* to the known constant address, otherwise abort.

This protection is *near-precise*, i.e. it guarantees that the control flow will be transferred to one of the valid return addresses for this function, but not necessary to the point of the most recent *CALL*.

General case

For functions called multiple times a *return jump table* is constructed in the same manner as jump tables for indirect function calls (see above). The correct jump table entry (or its index) is passed by *CALL* to *f()* (as an extra argument) and then spilled to stack. The *RET* instruction is replaced with a load of the jump table entry, jump table range check, and *JMP* to the jump table entry.

This protection is also *near-precise*.

Returns from functions called indirectly

If a function is called indirectly, the return jump table is constructed for the equivalence class of functions instead of a single function.

Cross-DSO calls

Consider two instrumented DSOs, *A* and *B*. *A* defines *f()* and *B* calls it.

This case will be handled similarly to the cross-DSO scheme using the slow path callback.

Non-goals

RCFI does not protect *RET* instructions:

- in non-instrumented DSOs,
- in instrumented DSOs for functions that are called from non-instrumented DSOs,
- embedded into other instructions (e.g. *0f4fc3 cmovg %ebx,%eax*).

Hardware support

We believe that the above design can be efficiently implemented in hardware. A single new instruction added to an ISA would allow to perform the forward-edge CFI check with fewer bytes per check (smaller code size overhead) and potentially more efficiently. The current software-only instrumentation requires at least 32-bytes per check (on x86_64). A hardware instruction may probably be less than ~ 12 bytes. Such instruction would check that the argument pointer is in-bounds, and is properly aligned, and if the checks fail it will either trap (in monolithic scheme) or call the slow path function (cross-DSO scheme). The bit vector lookup is probably too complex for a hardware implementation.

```
//  This instruction checks that 'Ptr'
//   * is aligned by (1 << kAlignment) and
//   * is inside [kRangeBeg, kRangeBeg+(kRangeSize<<kAlignment))
//  and if the check fails it jumps to the given target (slow path).
//
// 'Ptr' is a register, pointing to the virtual function table
//    or to the function which we need to check. We may require an explicit
//    fixed register to be used.
// 'kAlignment' is a 4-bit constant.
// 'kRangeSize' is a ~20-bit constant.
// 'kRangeBeg' is a PC-relative constant (~28 bits)
//    pointing to the beginning of the allowed range for 'Ptr'.
// 'kFailedCheckTarget': is a PC-relative constant (~28 bits)
//    representing the target to branch to when the check fails.
//    If kFailedCheckTarget==0, the process will trap
//    (monolithic binary scheme).
//    Otherwise it will jump to a handler that implements `CFI_SlowPath`
//    (cross-DSO scheme).
CFI_Check(Ptr, kAlignment, kRangeSize, kRangeBeg, kFailedCheckTarget) {
   if (Ptr < kRangeBeg ||
```

```
        Ptr >= kRangeBeg + (kRangeSize << kAlignment) ||
        Ptr & ((1 << kAlignment) - 1))
          Jump(kFailedCheckTarget);
}
```

An alternative and more compact encoding would not use *kFailedCheckTarget*, and will trap on check failure instead. This will allow us to fit the instruction into **8-9 bytes**. The cross-DSO checks will be performed by a trap handler and performance-critical ones will have to be black-listed and checked using the software-only scheme.

Note that such hardware extension would be complementary to checks at the callee side, such as e.g. **Intel ENDBRANCH**. Moreover, CFI would have two benefits over ENDBRANCH: a) precision and b) ability to protect against invalid casts between polymorphic types.

Introduction 684
Available schemes 685
Trapping and Diagnostics 685
Forward-Edge CFI for Virtual Calls 685
Performance 685
Bad Cast Checking 685
Non-Virtual Member Function Call Checking 686
Strictness 686
Indirect Function Call Checking 686
`-fsanitize-cfi-icall-generalize-pointers` 686
`-fsanitize-cfi-canonical-jump-tables` 687
`-fsanitize=cfi-icall` and `-fsanitize=function` 687
Member Function Pointer Call Checking 688
Ignorelist 688
Shared library support 688
Design 688
Publications 688

Introduction

Clang includes an implementation of a number of control flow integrity (CFI) schemes, which are designed to abort the program upon detecting certain forms of undefined behavior that can potentially allow attackers to subvert the program's control flow. These schemes have been optimized for performance, allowing developers to enable them in release builds.

To enable Clang's available CFI schemes, use the flag `-fsanitize=cfi`. You can also enable a subset of available schemes. As currently implemented, all schemes rely on link-time optimization (LTO); so it is required to specify `-flto`, and the linker used must support LTO, for example via the gold plugin.

To allow the checks to be implemented efficiently, the program must be structured such that certain object files are compiled with CFI enabled, and are statically linked into the program. This may preclude the use of shared libraries in some cases.

The compiler will only produce CFI checks for a class if it can infer hidden LTO visibility for that class. LTO visibility is a property of a class that is inferred from flags and attributes. For more details, see the documentation for LTO visibility.

The `-fsanitize=cfi-{vcall,nvcall,derived-cast,unrelated-cast}` flags require that a `-fvisibility=` flag also be specified. This is because the default visibility setting is `-fvisibility=default`, which would disable CFI checks for classes without visibility attributes. Most users will want to specify `-fvisibility=hidden`, which enables CFI checks for such classes.

Experimental support for cross-DSO control flow integrity exists that does not require classes to have hidden LTO visibility. This cross-DSO support has unstable ABI at this time.

Available schemes

Available schemes are:

- `-fsanitize=cfi-cast-strict`: Enables strict cast checks.
- `-fsanitize=cfi-derived-cast`: Base-to-derived cast to the wrong dynamic type.
- `-fsanitize=cfi-unrelated-cast`: Cast from `void*` or another unrelated type to the wrong dynamic type.
- `-fsanitize=cfi-nvcall`: Non-virtual call via an object whose vptr is of the wrong dynamic type.
- `-fsanitize=cfi-vcall`: Virtual call via an object whose vptr is of the wrong dynamic type.
- `-fsanitize=cfi-icall`: Indirect call of a function with wrong dynamic type.
- `-fsanitize=cfi-mfcall`: Indirect call via a member function pointer with wrong dynamic type.

You can use `-fsanitize=cfi` to enable all the schemes and use `-fno-sanitize` flag to narrow down the set of schemes as desired. For example, you can build your program with `-fsanitize=cfi -fno-sanitize=cfi-nvcall,cfi-icall` to use all schemes except for non-virtual member function call and indirect call checking.

Remember that you have to provide `-flto` or `-flto=thin` if at least one CFI scheme is enabled.

Trapping and Diagnostics

By default, CFI will abort the program immediately upon detecting a control flow integrity violation. You can use the -fno-sanitize-trap= flag to cause CFI to print a diagnostic similar to the one below before the program aborts.

```
bad-cast.cpp:109:7: runtime error: control flow integrity check for type 'B' failed during base-to-derived cast (vtable address 0x000000425a50)
0x000000425a50: note: vtable is of type 'A'
 00 00 00 00  f0 f1 41 00 00 00 00 00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  20 5a 42 00
              ^
```

If diagnostics are enabled, you can also configure CFI to continue program execution instead of aborting by using the -fsanitize-recover= flag.

Forward-Edge CFI for Virtual Calls

This scheme checks that virtual calls take place using a vptr of the correct dynamic type; that is, the dynamic type of the called object must be a derived class of the static type of the object used to make the call. This CFI scheme can be enabled on its own using `-fsanitize=cfi-vcall`.

For this scheme to work, all translation units containing the definition of a virtual member function (whether inline or not), other than members of ignored types or types with public LTO visibility, must be compiled with `-flto` or `-flto=thin` enabled and be statically linked into the program.

Performance

A performance overhead of less than 1% has been measured by running the Dromaeo benchmark suite against an instrumented version of the Chromium web browser. Another good performance benchmark for this mechanism is the virtual-call-heavy SPEC 2006 xalancbmk.

Note that this scheme has not yet been optimized for binary size; an increase of up to 15% has been observed for Chromium.

Bad Cast Checking

This scheme checks that pointer casts are made to an object of the correct dynamic type; that is, the dynamic type of the object must be a derived class of the pointee type of the cast. The checks are currently only introduced where the class being casted to is a polymorphic class.

Bad casts are not in themselves control flow integrity violations, but they can also create security vulnerabilities, and the implementation uses many of the same mechanisms.

There are two types of bad cast that may be forbidden: bad casts from a base class to a derived class (which can be checked with `-fsanitize=cfi-derived-cast`), and bad casts from a pointer of type `void*` or another unrelated type (which can be checked with `-fsanitize=cfi-unrelated-cast`).

The difference between these two types of casts is that the first is defined by the C++ standard to produce an undefined value, while the second is not in itself undefined behavior (it is well defined to cast the pointer back to its original type) unless the object is uninitialized and the cast is a `static_cast` (see C++14 [basic.life]p5).

If a program as a matter of policy forbids the second type of cast, that restriction can normally be enforced. However it may in some cases be necessary for a function to perform a forbidden cast to conform with an external API (e.g. the `allocate` member function of a standard library allocator). Such functions may be ignored.

For this scheme to work, all translation units containing the definition of a virtual member function (whether inline or not), other than members of ignored types or types with public LTO visibility, must be compiled with `-flto` or `-flto=thin` enabled and be statically linked into the program.

Non-Virtual Member Function Call Checking

This scheme checks that non-virtual calls take place using an object of the correct dynamic type; that is, the dynamic type of the called object must be a derived class of the static type of the object used to make the call. The checks are currently only introduced where the object is of a polymorphic class type. This CFI scheme can be enabled on its own using `-fsanitize=cfi-nvcall`.

For this scheme to work, all translation units containing the definition of a virtual member function (whether inline or not), other than members of ignored types or types with public LTO visibility, must be compiled with `-flto` or `-flto=thin` enabled and be statically linked into the program.

Strictness

If a class has a single non-virtual base and does not introduce or override virtual member functions or fields other than an implicitly defined virtual destructor, it will have the same layout and virtual function semantics as its base. By default, casts to such classes are checked as if they were made to the least derived such class.

Casting an instance of a base class to such a derived class is technically undefined behavior, but it is a relatively common hack for introducing member functions on class instances with specific properties that works under most compilers and should not have security implications, so we allow it by default. It can be disabled with `-fsanitize=cfi-cast-strict`.

Indirect Function Call Checking

This scheme checks that function calls take place using a function of the correct dynamic type; that is, the dynamic type of the function must match the static type used at the call. This CFI scheme can be enabled on its own using `-fsanitize=cfi-icall`.

For this scheme to work, each indirect function call in the program, other than calls in ignored functions, must call a function which was either compiled with `-fsanitize=cfi-icall` enabled, or whose address was taken by a function in a translation unit compiled with `-fsanitize=cfi-icall`.

If a function in a translation unit compiled with `-fsanitize=cfi-icall` takes the address of a function not compiled with `-fsanitize=cfi-icall`, that address may differ from the address taken by a function in a translation unit not compiled with `-fsanitize=cfi-icall`. This is technically a violation of the C and C++ standards, but it should not affect most programs.

Each translation unit compiled with `-fsanitize=cfi-icall` must be statically linked into the program or shared library, and calls across shared library boundaries are handled as if the callee was not compiled with `-fsanitize=cfi-icall`.

This scheme is currently supported on a limited set of targets: x86, x86_64, arm, arch64 and wasm.

`-fsanitize-cfi-icall-generalize-pointers`

Mismatched pointer types are a common cause of cfi-icall check failures. Translation units compiled with the `-fsanitize-cfi-icall-generalize-pointers` flag relax pointer type checking for call sites in that translation unit, applied across all functions compiled with `-fsanitize=cfi-icall`.

Specifically, pointers in return and argument types are treated as equivalent as long as the qualifiers for the type they point to match. For example, `char*`, `char**`, and `int*` are considered equivalent types. However, `char*` and `const char*` are considered separate types.

`-fsanitize-cfi-icall-generalize-pointers` is not compatible with `-fsanitize-cfi-cross-dso`.

`-fsanitize-cfi-canonical-jump-tables`

The default behavior of Clang's indirect function call checker will replace the address of each CFI-checked function in the output file's symbol table with the address of a jump table entry which will pass CFI checks. We refer to this as making the jump table *canonical*. This property allows code that was not compiled with `-fsanitize=cfi-icall` to take a CFI-valid address of a function, but it comes with a couple of caveats that are especially relevant for users of cross-DSO CFI:

- There is a performance and code size overhead associated with each exported function, because each such function must have an associated jump table entry, which must be emitted even in the common case where the function is never address-taken anywhere in the program, and must be used even for direct calls between DSOs, in addition to the PLT overhead.
- There is no good way to take a CFI-valid address of a function written in assembly or a language not supported by Clang. The reason is that the code generator would need to insert a jump table in order to form a CFI-valid address for assembly functions, but there is no way in general for the code generator to determine the language of the function. This may be possible with LTO in the intra-DSO case, but in the cross-DSO case the only information available is the function declaration. One possible solution is to add a C wrapper for each assembly function, but these wrappers can present a significant maintenance burden for heavy users of assembly in addition to adding runtime overhead.

For these reasons, we provide the option of making the jump table non-canonical with the flag `-fno-sanitize-cfi-canonical-jump-tables`. When the jump table is made non-canonical, symbol table entries point directly to the function body. Any instances of a function's address being taken in C will be replaced with a jump table address.

This scheme does have its own caveats, however. It does end up breaking function address equality more aggressively than the default behavior, especially in cross-DSO mode which normally preserves function address equality entirely.

Furthermore, it is occasionally necessary for code not compiled with `-fsanitize=cfi-icall` to take a function address that is valid for CFI. For example, this is necessary when a function's address is taken by assembly code and then called by CFI-checking C code. The `__attribute__((cfi_canonical_jump_table))` attribute may be used to make the jump table entry of a specific function canonical so that the external code will end up taking an address for the function that will pass CFI checks.

`-fsanitize=cfi-icall` and `-fsanitize=function`

This tool is similar to `-fsanitize=function` in that both tools check the types of function calls. However, the two tools occupy different points on the design space; `-fsanitize=function` is a developer tool designed to find bugs in local development builds, whereas `-fsanitize=cfi-icall` is a security hardening mechanism designed to be deployed in release builds.

`-fsanitize=function` has a higher space and time overhead due to a more complex type check at indirect call sites, as well as a need for run-time type information (RTTI), which may make it unsuitable for deployment. Because of the need for RTTI, `-fsanitize=function` can only be used with C++ programs, whereas `-fsanitize=cfi-icall` can protect both C and C++ programs.

On the other hand, `-fsanitize=function` conforms more closely with the C++ standard and user expectations around interaction with shared libraries; the identity of function pointers is maintained, and calls across shared library boundaries are no different from calls within a single program or shared library.

Member Function Pointer Call Checking

This scheme checks that indirect calls via a member function pointer take place using an object of the correct dynamic type. Specifically, we check that the dynamic type of the member function referenced by the member function pointer matches the “function pointer” part of the member function pointer, and that the member function’s class type is related to the base type of the member function. This CFI scheme can be enabled on its own using `-fsanitize=cfi-mfcall`.

The compiler will only emit a full CFI check if the member function pointer’s base type is complete. This is because the complete definition of the base type contains information that is necessary to correctly compile the CFI check. To ensure that the compiler always emits a full CFI check, it is recommended to also pass the flag `-fcomplete-member-pointers`, which enables a non-conforming language extension that requires member pointer base types to be complete if they may be used for a call.

For this scheme to work, all translation units containing the definition of a virtual member function (whether inline or not), other than members of ignored types or types with public LTO visibility, must be compiled with `-flto` or `-flto=thin` enabled and be statically linked into the program.

This scheme is currently not compatible with cross-DSO CFI or the Microsoft ABI.

Ignorelist

A Sanitizer special case list can be used to relax CFI checks for certain source files, functions and types using the `src`, `fun` and `type` entity types. Specific CFI modes can be be specified using `[section]` headers.

```
# Suppress all CFI checking for code in a file.
src:bad_file.cpp
src:bad_header.h
# Ignore all functions with names containing MyFooBar.
fun:*MyFooBar*
# Ignore all types in the standard library.
type:std::*
# Disable only unrelated cast checks for this function
[cfi-unrelated-cast]
fun:*UnrelatedCast*
# Disable CFI call checks for this function without affecting cast checks
[cfi-vcall|cfi-nvcall|cfi-icall]
fun:*BadCall*
```

Shared library support

Use **-f[no-]sanitize-cfi-cross-dso** to enable the cross-DSO control flow integrity mode, which allows all CFI schemes listed above to apply across DSO boundaries. As in the regular CFI, each DSO must be built with `-flto` or `-flto=thin`.

Normally, CFI checks will only be performed for classes that have hidden LTO visibility. With this flag enabled, the compiler will emit cross-DSO CFI checks for all classes, except for those which appear in the CFI ignorelist or which use a `no_sanitize` attribute.

Design

Please refer to the design document.

Publications

Control-Flow Integrity: Principles, Implementations, and Applications. Martin Abadi, Mihai Budiu, Úlfar Erlingsson, Jay Ligatti.

Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, Geoff Pike.

LTO Visibility

LTO visibility is a property of an entity that specifies whether it can be referenced from outside the current LTO unit. A *linkage unit* is a set of translation units linked together into an executable or DSO, and a linkage unit's *LTO unit* is the subset of the linkage unit that is linked together using link-time optimization; in the case where LTO is not being used, the linkage unit's LTO unit is empty. Each linkage unit has only a single LTO unit.

The LTO visibility of a class is used by the compiler to determine which classes the whole-program devirtualization (`-fwhole-program-vtables`) and control flow integrity (`-fsanitize=cfi-vcall` and `-fsanitize=cfi-mfcall`) features apply to. These features use whole-program information, so they require the entire class hierarchy to be visible in order to work correctly.

If any translation unit in the program uses either of the whole-program devirtualization or control flow integrity features, it is effectively an ODR violation to define a class with hidden LTO visibility in multiple linkage units. A class with public LTO visibility may be defined in multiple linkage units, but the tradeoff is that the whole-program devirtualization and control flow integrity features can only be applied to classes with hidden LTO visibility. A class's LTO visibility is treated as an ODR-relevant property of its definition, so it must be consistent between translation units.

In translation units built with LTO, LTO visibility is based on the class's symbol visibility as expressed at the source level (i.e. the `__attribute__((visibility("...")))` attribute, or the `-fvisibility=` flag) or, on the Windows platform, the dllimport and dllexport attributes. When targeting non-Windows platforms, classes with a visibility other than hidden visibility receive public LTO visibility. When targeting Windows, classes with dllimport or dllexport attributes receive public LTO visibility. All other classes receive hidden LTO visibility. Classes with internal linkage (e.g. classes declared in unnamed namespaces) also receive hidden LTO visibility.

During the LTO link, all classes with public LTO visibility will be refined to hidden LTO visibility when the `--lto-whole-program-visibility` lld linker option is applied (`-plugin-opt=whole-program-visibility` for gold). This flag can be used to defer specifying whether classes have hidden LTO visibility until link time, to allow bitcode objects to be shared by different LTO links. Due to an implementation limitation, symbols associated with classes with hidden LTO visibility may still be exported from the binary when using this flag. It is unsafe to refer to these symbols, and their visibility may be relaxed to hidden in a future compiler release.

A class defined in a translation unit built without LTO receives public LTO visibility regardless of its object file visibility, linkage or other attributes.

This mechanism will produce the correct result in most cases, but there are two cases where it may wrongly infer hidden LTO visibility.

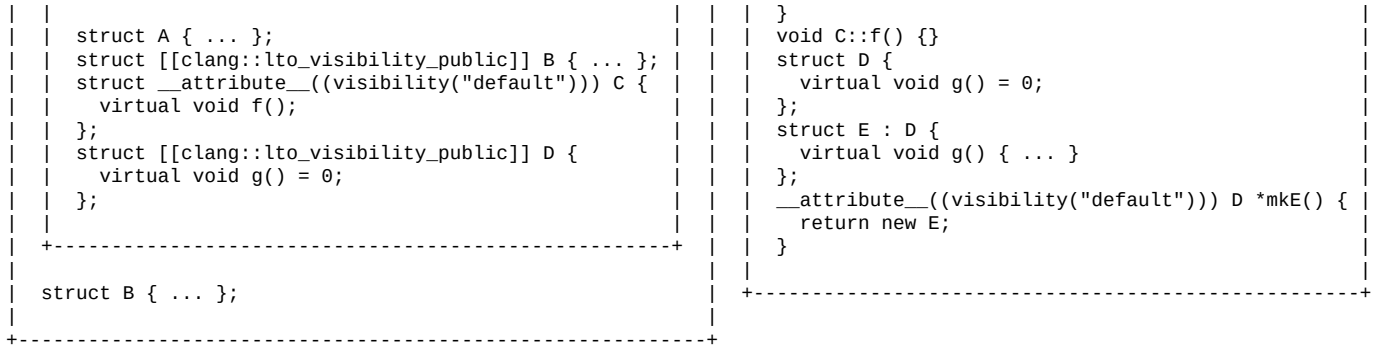
1. As a corollary of the above rules, if a linkage unit is produced from a combination of LTO object files and non-LTO object files, any hidden visibility class defined in both a translation unit built with LTO and a translation unit built without LTO must be defined with public LTO visibility in order to avoid an ODR violation.
2. Some ABIs provide the ability to define an abstract base class without visibility attributes in multiple linkage units and have virtual calls to derived classes in other linkage units work correctly. One example of this is COM on Windows platforms. If the ABI allows this, any base class used in this way must be defined with public LTO visibility.

Classes that fall into either of these categories can be marked up with the `[[clang::lto_visibility_public]]` attribute. To specifically handle the COM case, classes with the `__declspec(uuid())` attribute receive public LTO visibility. On Windows platforms, clang-cl's `/MT` and `/MTd` flags statically link the program against a prebuilt standard library; these flags imply public LTO visibility for every class declared in the `std` and `stdext` namespaces.

Example

The following example shows how LTO visibility works in practice in several cases involving two linkage units, `main` and `dso.so`.

```
+-----------------------------------------------------------+  +-------------------------------------------------+
| main (clang++ -fvisibility=hidden):                       |  | dso.so (clang++ -fvisibility=hidden):           |
|                                                           |  |                                                 |
|  +-----------------------------------------------------+  |  |  struct __attribute__((visibility("default"))) C { |
|  | LTO unit (clang++ -fvisibility=hidden -flto):       |  |  |    virtual void f();                            |
```



We will now describe the LTO visibility of each of the classes defined in these linkage units.

Class `A` is not defined outside of `main`'s LTO unit, so it can have hidden LTO visibility. This is inferred from the object file visibility specified on the command line.

Class `B` is defined in `main`, both inside and outside its LTO unit. The definition outside the LTO unit has public LTO visibility, so the definition inside the LTO unit must also have public LTO visibility in order to avoid an ODR violation.

Class `C` is defined in both `main` and `dso.so` and therefore must have public LTO visibility. This is correctly inferred from the `visibility` attribute.

Class `D` is an abstract base class with a derived class `E` defined in `dso.so`. This is an example of the COM scenario; the definition of `D` in `main`'s LTO unit must have public LTO visibility in order to be compatible with the definition of `D` in `dso.so`, which is observable by calling the function `mkE`.

SafeStack

Introduction **690**
Performance 691
Compatibility 691
Known compatibility limitations 691
Security 691
Known security limitations 691
Usage **692**
Supported Platforms 692
Low-level API 692
`__has_feature(safe_stack)` 692
`__attribute__((no_sanitize("safe-stack")))` 692
`__builtin___get_unsafe_stack_ptr()` 692
`__builtin___get_unsafe_stack_bottom()` 692
`__builtin___get_unsafe_stack_top()` 692
`__builtin___get_unsafe_stack_start()` 692
Design **692**
setjmp and exception handling 693
Publications 693

Introduction

SafeStack is an instrumentation pass that protects programs against attacks based on stack buffer overflows, without introducing any measurable performance overhead. It works by separating the program stack into two distinct regions: the safe stack and the unsafe stack. The safe stack stores return addresses, register spills, and local variables that are always accessed in a safe way, while the unsafe stack stores everything else. This separation ensures that buffer overflows on the unsafe stack cannot be used to overwrite anything on the safe stack.

SafeStack is a part of the Code-Pointer Integrity (CPI) Project.

Performance

The performance overhead of the SafeStack instrumentation is less than 0.1% on average across a variety of benchmarks (see the Code-Pointer Integrity paper for details). This is mainly because most small functions do not have any variables that require the unsafe stack and, hence, do not need unsafe stack frames to be created. The cost of creating unsafe stack frames for large functions is amortized by the cost of executing the function.

In some cases, SafeStack actually improves the performance. Objects that end up being moved to the unsafe stack are usually large arrays or variables that are used through multiple stack frames. Moving such objects away from the safe stack increases the locality of frequently accessed values on the stack, such as register spills, return addresses, and small local variables.

Compatibility

Most programs, static libraries, or individual files can be compiled with SafeStack as is. SafeStack requires basic runtime support, which, on most platforms, is implemented as a compiler-rt library that is automatically linked in when the program is compiled with SafeStack.

Linking a DSO with SafeStack is not currently supported.

Known compatibility limitations

Certain code that relies on low-level stack manipulations requires adaption to work with SafeStack. One example is mark-and-sweep garbage collection implementations for C/C++ (e.g., Oilpan in chromium/blink), which must be changed to look for the live pointers on both safe and unsafe stacks.

SafeStack supports linking statically modules that are compiled with and without SafeStack. An executable compiled with SafeStack can load dynamic libraries that are not compiled with SafeStack. At the moment, compiling dynamic libraries with SafeStack is not supported.

Signal handlers that use `sigaltstack()` must not use the unsafe stack (see `__attribute__((no_sanitize("safe-stack")))` below).

Programs that use APIs from `ucontext.h` are not supported yet.

Security

SafeStack protects return addresses, spilled registers and local variables that are always accessed in a safe way by separating them in a dedicated safe stack region. The safe stack is automatically protected against stack-based buffer overflows, since it is disjoint from the unsafe stack in memory, and it itself is always accessed in a safe way. In the current implementation, the safe stack is protected against arbitrary memory write vulnerabilities though randomization and information hiding: the safe stack is allocated at a random address and the instrumentation ensures that no pointers to the safe stack are ever stored outside of the safe stack itself (see limitations below).

Known security limitations

A complete protection against control-flow hijack attacks requires combining SafeStack with another mechanism that enforces the integrity of code pointers that are stored on the heap or the unsafe stack, such as CPI, or a forward-edge control flow integrity mechanism that enforces correct calling conventions at indirect call sites, such as IFCC with arity checks. Clang has control-flow integrity protection scheme for C++ virtual calls, but not non-virtual indirect calls. With SafeStack alone, an attacker can overwrite a function pointer on the heap or the unsafe stack and cause a program to call arbitrary location, which in turn might enable stack pivoting and return-oriented programming.

In its current implementation, SafeStack provides precise protection against stack-based buffer overflows, but protection against arbitrary memory write vulnerabilities is probabilistic and relies on randomization and information hiding. The randomization is currently based on system-enforced ASLR and shares its known security limitations. The safe stack pointer hiding is not perfect yet either: system library functions such as `swapcontext`, exception handling mechanisms, intrinsics such as `__builtin_frame_address`, or low-level bugs in runtime support could leak the safe stack pointer. In the future, such leaks could be detected by static or dynamic analysis tools and prevented by adjusting such functions to either encrypt the stack pointer when storing it in the heap (as already done e.g., by `setjmp`/`longjmp` implementation in glibc), or store it in a safe region instead.

The CPI paper describes two alternative, stronger safe stack protection mechanisms, that rely on software fault isolation, or hardware segmentation (as available on x86-32 and some x86-64 CPUs).

At the moment, SafeStack assumes that the compiler's implementation is correct. This has not been verified except through manual code inspection, and could always regress in the future. It's therefore desirable to have a separate static or dynamic binary verification tool that would check the correctness of the SafeStack instrumentation in final binaries.

Usage

To enable SafeStack, just pass `-fsanitize=safe-stack` flag to both compile and link command lines.

Supported Platforms

SafeStack was tested on Linux, NetBSD, FreeBSD and macOS.

Low-level API

`__has_feature(safe_stack)`

In some rare cases one may need to execute different code depending on whether SafeStack is enabled. The macro `__has_feature(safe_stack)` can be used for this purpose.

```
#if __has_feature(safe_stack)
// code that builds only under SafeStack
#endif
```

`__attribute__((no_sanitize("safe-stack")))`

Use `__attribute__((no_sanitize("safe-stack")))` on a function declaration to specify that the safe stack instrumentation should not be applied to that function, even if enabled globally (see `-fsanitize=safe-stack` flag). This attribute may be required for functions that make assumptions about the exact layout of their stack frames.

All local variables in functions with this attribute will be stored on the safe stack. The safe stack remains unprotected against memory errors when accessing these variables, so extra care must be taken to manually ensure that all such accesses are safe. Furthermore, the addresses of such local variables should never be stored on the heap, as it would leak the location of the SafeStack.

`__builtin___get_unsafe_stack_ptr()`

This builtin function returns current unsafe stack pointer of the current thread.

`__builtin___get_unsafe_stack_bottom()`

This builtin function returns a pointer to the bottom of the unsafe stack of the current thread.

`__builtin___get_unsafe_stack_top()`

This builtin function returns a pointer to the top of the unsafe stack of the current thread.

`__builtin___get_unsafe_stack_start()`

Deprecated: This builtin function is an alias for `__builtin___get_unsafe_stack_bottom()`.

Design

Please refer to the Code-Pointer Integrity project page for more information about the design of the SafeStack and its related technologies.

setjmp and exception handling

The OSDI'14 paper mentions that on Linux the instrumentation pass finds calls to setjmp or functions that may throw an exception, and inserts required instrumentation at their call sites. Specifically, the instrumentation pass saves the shadow stack pointer on the safe stack before the call site, and restores it either after the call to setjmp or after an exception has been caught. This is implemented in the function `SafeStack::createStackRestorePoints`.

Publications

Code-Pointer Integrity. Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, Dawn Song. USENIX Symposium on Operating Systems Design and Implementation (OSDI), Broomfield, CO, October 2014

ShadowCallStack

Introduction **693**
Comparison 693
Compatibility 693
Security **694**
Usage **694**
Low-level API 695
`__has_feature(shadow_call_stack)` 695
`__attribute__((no_sanitize("shadow-call-stack")))` 695
Example **695**

Introduction

ShadowCallStack is an instrumentation pass, currently only implemented for aarch64, that protects programs against return address overwrites (e.g. stack buffer overflows.) It works by saving a function's return address to a separately allocated 'shadow call stack' in the function prolog in non-leaf functions and loading the return address from the shadow call stack in the function epilog. The return address is also stored on the regular stack for compatibility with unwinders, but is otherwise unused.

The aarch64 implementation is considered production ready, and an implementation of the runtime has been added to Android's libc (bionic). An x86_64 implementation was evaluated using Chromium and was found to have critical performance and security deficiencies–it was removed in LLVM 9.0. Details on the x86_64 implementation can be found in the Clang 7.0.1 documentation.

Comparison

To optimize for memory consumption and cache locality, the shadow call stack stores only an array of return addresses. This is in contrast to other schemes, like SafeStack, that mirror the entire stack and trade-off consuming more memory for shorter function prologs and epilogs with fewer memory accesses.

Return Flow Guard is a pure software implementation of shadow call stacks on x86_64. Like the previous implementation of ShadowCallStack on x86_64, it is inherently racy due to the architecture's use of the stack for calls and returns.

Intel Control-flow Enforcement Technology (CET) is a proposed hardware extension that would add native support to use a shadow stack to store/check return addresses at call/return time. Being a hardware implementation, it would not suffer from race conditions and would not incur the overhead of function instrumentation, but it does require operating system support.

Compatibility

A runtime is not provided in compiler-rt so one must be provided by the compiled application or the operating system. Integrating the runtime into the operating system should be preferred since otherwise all thread creation and destruction would need to be intercepted by the application.

The instrumentation makes use of the platform register `x18`. On some platforms, `x18` is reserved, and on others, it is designated as a scratch register. This generally means that any code that may run on the same thread as code compiled with ShadowCallStack must either target one of the platforms whose ABI reserves `x18` (currently Android, Darwin, Fuchsia and Windows) or be compiled with the flag `-ffixed-x18`. If absolutely necessary, code compiled without `-ffixed-x18` may be run on the same thread as code that uses ShadowCallStack by saving the register value temporarily on the stack (example in Android) but this should be done with care since it risks leaking the shadow call stack address.

Because of the use of register `x18`, the ShadowCallStack feature is incompatible with any other feature that may use `x18`. However, there is no inherent reason why ShadowCallStack needs to use register `x18` specifically; in principle, a platform could choose to reserve and use another register for ShadowCallStack, but this would be incompatible with the AAPCS64.

Special unwind information is required on functions that are compiled with ShadowCallStack and that may be unwound, i.e. functions compiled with `-fexceptions` (which is the default in C++). Some unwinders (such as the libgcc 4.9 unwinder) do not understand this unwind info and will segfault when encountering it. LLVM libunwind processes this unwind info correctly, however. This means that if exceptions are used together with ShadowCallStack, the program must use a compatible unwinder.

Security

ShadowCallStack is intended to be a stronger alternative to `-fstack-protector`. It protects from non-linear overflows and arbitrary memory writes to the return address slot.

The instrumentation makes use of the `x18` register to reference the shadow call stack, meaning that references to the shadow call stack do not have to be stored in memory. This makes it possible to implement a runtime that avoids exposing the address of the shadow call stack to attackers that can read arbitrary memory. However, attackers could still try to exploit side channels exposed by the operating system [1] [2] or processor [3] to discover the address of the shadow call stack.

Unless care is taken when allocating the shadow call stack, it may be possible for an attacker to guess its address using the addresses of other allocations. Therefore, the address should be chosen to make this difficult. One way to do this is to allocate a large guard region without read/write permissions, randomly select a small region within it to be used as the address of the shadow call stack and mark only that region as read/write. This also mitigates somewhat against processor side channels. The intent is that the Android runtime will do this, but the platform will first need to be changed to avoid using `setrlimit(RLIMIT_AS)` to limit memory allocations in certain processes, as this also limits the number of guard regions that can be allocated.

The runtime will need the address of the shadow call stack in order to deallocate it when destroying the thread. If the entire program is compiled with `-ffixed-x18`, this is trivial: the address can be derived from the value stored in `x18` (e.g. by masking out the lower bits). If a guard region is used, the address of the start of the guard region could then be stored at the start of the shadow call stack itself. But if it is possible for code compiled without `-ffixed-x18` to run on a thread managed by the runtime, which is the case on Android for example, the address must be stored somewhere else instead. On Android we store the address of the start of the guard region in TLS and deallocate the entire guard region including the shadow call stack at thread exit. This is considered acceptable given that the address of the start of the guard region is already somewhat guessable.

One way in which the address of the shadow call stack could leak is in the `jmp_buf` data structure used by `setjmp` and `longjmp`. The Android runtime avoids this by only storing the low bits of `x18` in the `jmp_buf`, which requires the address of the shadow call stack to be aligned to its size.

The architecture's call and return instructions (`bl` and `ret`) operate on a register rather than the stack, which means that leaf functions are generally protected from return address overwrites even without ShadowCallStack.

Usage

To enable ShadowCallStack, just pass the `-fsanitize=shadow-call-stack` flag to both compile and link command lines. On aarch64, you also need to pass `-ffixed-x18` unless your target already reserves `x18`.

Low-level API

`__has_feature(shadow_call_stack)`

In some cases one may need to execute different code depending on whether ShadowCallStack is enabled. The macro `__has_feature(shadow_call_stack)` can be used for this purpose.

```
#if defined(__has_feature)
#  if __has_feature(shadow_call_stack)
// code that builds only under ShadowCallStack
#  endif
#endif
```

`__attribute__((no_sanitize("shadow-call-stack")))`

Use `__attribute__((no_sanitize("shadow-call-stack")))` on a function declaration to specify that the shadow call stack instrumentation should not be applied to that function, even if enabled globally.

Example

The following example code:

```
int foo() {
  return bar() + 1;
}
```

Generates the following aarch64 assembly when compiled with `-O2`:

```
stp     x29, x30, [sp, #-16]!
mov     x29, sp
bl      bar
add     w0, w0, #1
ldp     x29, x30, [sp], #16
ret
```

Adding `-fsanitize=shadow-call-stack` would output the following assembly:

```
str     x30, [x18], #8
stp     x29, x30, [sp, #-16]!
mov     x29, sp
bl      bar
add     w0, w0, #1
ldp     x29, x30, [sp], #16
ldr     x30, [x18, #-8]!
ret
```

Source-based Code Coverage

Introduction 696

The code coverage workflow 696

Compiling with coverage enabled 697

Running the instrumented program 697

Creating coverage reports 697

Exporting coverage data 699

Interpreting reports 699

Format compatibility guarantees 699

Impact of llvm optimizations on coverage reports 700

Using the profiling runtime without static initializers 700

Using the profiling runtime without a filesystem 700

Collecting coverage reports for the llvm project 700

Drawbacks and limitations 700

Clang implementation details 701

Gap regions 701

Branch regions 701

Switch statements 701

Introduction

This document explains how to use clang's source-based code coverage feature. It's called "source-based" because it operates on AST and preprocessor information directly. This allows it to generate very precise coverage data.

Clang ships two other code coverage implementations:

- SanitizerCoverage - A low-overhead tool meant for use alongside the various sanitizers. It can provide up to edge-level coverage.
- gcov - A GCC-compatible coverage implementation which operates on DebugInfo. This is enabled by `-ftest-coverage` or `--coverage`.

From this point onwards "code coverage" will refer to the source-based kind.

The code coverage workflow

The code coverage workflow consists of three main steps:

- Compiling with coverage enabled.
- Running the instrumented program.
- Creating coverage reports.

The next few sections work through a complete, copy-'n-paste friendly example based on this program:

```
% cat <<EOF > foo.cc
#define BAR(x) ((x) || (x))
template <typename T> void foo(T x) {
  for (unsigned I = 0; I < 10; ++I) { BAR(I); }
}
int main() {
  foo<int>(0);
  foo<float>(0);
  return 0;
}
EOF
```

Compiling with coverage enabled

To compile code with coverage enabled, pass `-fprofile-instr-generate -fcoverage-mapping` to the compiler:

```
# Step 1: Compile with coverage enabled.
% clang++ -fprofile-instr-generate -fcoverage-mapping foo.cc -o foo
```

Note that linking together code with and without coverage instrumentation is supported. Uninstrumented code simply won't be accounted for in reports.

Running the instrumented program

The next step is to run the instrumented program. When the program exits it will write a **raw profile** to the path specified by the `LLVM_PROFILE_FILE` environment variable. If that variable does not exist, the profile is written to `default.profraw` in the current directory of the program. If `LLVM_PROFILE_FILE` contains a path to a non-existent directory, the missing directory structure will be created. Additionally, the following special **pattern strings** are rewritten:

- "%p" expands out to the process ID.
- "%h" expands out to the hostname of the machine running the program.
- "%t" expands out to the value of the `TMPDIR` environment variable. On Darwin, this is typically set to a temporary scratch directory.
- "%Nm" expands out to the instrumented binary's signature. When this pattern is specified, the runtime creates a pool of N raw profiles which are used for on-line profile merging. The runtime takes care of selecting a raw profile from the pool, locking it, and updating it before the program exits. If N is not specified (i.e the pattern is "%m"), it's assumed that `N = 1`. N must be between 1 and 9. The merge pool specifier can only occur once per filename pattern.
- "%c" expands out to nothing, but enables a mode in which profile counter updates are continuously synced to a file. This means that if the instrumented program crashes, or is killed by a signal, perfect coverage information can still be recovered. Continuous mode does not support value profiling for PGO, and is only supported on Darwin at the moment. Support for Linux may be mostly complete but requires testing, and support for Windows may require more extensive changes: please get involved if you are interested in porting this feature.

```
# Step 2: Run the program.
% LLVM_PROFILE_FILE="foo.profraw" ./foo
```

Note that continuous mode is also used on Fuchsia where it's the only supported mode, but the implementation is different. The Darwin and Linux implementation relies on padding and the ability to map a file over the existing memory mapping which is generally only available on POSIX systems and isn't suitable for other platforms.

On Fuchsia, we rely on the ability to relocate counters at runtime using a level of indirection. On every counter access, we add a bias to the counter address. This bias is stored in `__llvm_profile_counter_bias` symbol that's provided by the profile runtime and is initially set to zero, meaning no relocation. The runtime can map the profile into memory at arbitrary locations, and set bias to the offset between the original and the new counter location, at which point every subsequent counter access will be to the new location, which allows updating profile directly akin to the continuous mode.

The advantage of this approach is that doesn't require any special OS support. The disadvantage is the extra overhead due to additional instructions required for each counter access (overhead both in terms of binary size and performance) plus duplication of counters (i.e. one copy in the binary itself and another copy that's mapped into memory). This implementation can be also enabled for other platforms by passing the `-runtime-counter-relocation` option to the backend during compilation.

```
% clang++ -fprofile-instr-generate -fcoverage-mapping -mllvm -runtime-counter-relocation foo.cc -o foo
```

Creating coverage reports

Raw profiles have to be **indexed** before they can be used to generate coverage reports. This is done using the "merge" tool in `llvm-profdata` (which can combine multiple raw profiles and index them at the same time):

```
# Step 3(a): Index the raw profile.
% llvm-profdata merge -sparse foo.profraw -o foo.profdata
```

There are multiple different ways to render coverage reports. The simplest option is to generate a line-oriented report:

```
# Step 3(b): Create a line-oriented coverage report.
% llvm-cov show ./foo -instr-profile=foo.profdata
```

This report includes a summary view as well as dedicated sub-views for templated functions and their instantiations. For our example program, we get distinct views for `foo<int>(...)` and `foo<float>(...)`. If `-show-line-counts-or-regions` is enabled, `llvm-cov` displays sub-line region counts (even in macro expansions):

```
    1|   20|#define BAR(x) ((x) || (x))
                          ^20     ^2
    2|    2|template <typename T> void foo(T x) {
    3|   22|  for (unsigned I = 0; I < 10; ++I) { BAR(I); }
                                   ^22     ^20  ^20^20
    4|    2|}
------------------
| void foo<int>(int):
|      2|    1|template <typename T> void foo(T x) {
|      3|   11|  for (unsigned I = 0; I < 10; ++I) { BAR(I); }
|                                     ^11     ^10  ^10^10
|      4|    1|}
------------------
| void foo<float>(int):
|      2|    1|template <typename T> void foo(T x) {
|      3|   11|  for (unsigned I = 0; I < 10; ++I) { BAR(I); }
|                                     ^11     ^10  ^10^10
|      4|    1|}
------------------
```

If `--show-branches=count` and `--show-expansions` are also enabled, the sub-views will show detailed branch coverage information in addition to the region counts:

```
------------------
| void foo<float>(int):
|      2|    1|template <typename T> void foo(T x) {
|      3|   11|  for (unsigned I = 0; I < 10; ++I) { BAR(I); }
|                                     ^11     ^10  ^10^10
|  ------------------
|  |  |    1|     10|#define BAR(x) ((x) || (x))
|  |  |                            ^10     ^1
|  |  |  ------------------
|  |  |  |  Branch (1:17): [True: 9, False: 1]
|  |  |  |  Branch (1:24): [True: 0, False: 1]
|  |  |  ------------------
|  ------------------
|  |  Branch (3:23): [True: 10, False: 1]
|  ------------------
|      4|    1|}
------------------
```

To generate a file-level summary of coverage statistics instead of a line-oriented report, try:

```
# Step 3(c): Create a coverage summary.
% llvm-cov report ./foo -instr-profile=foo.profdata
Filename          Regions    Missed Regions     Cover   Functions  Missed Functions  Executed       Lines      Missed Lines     Cover     Branches    Missed Branches     Cover
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
/tmp/foo.cc            13                 0   100.00%           3                 0   100.00%          13                 0   100.00%          12                 2    83.33%
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
TOTAL                  13                 0   100.00%           3                 0   100.00%          13                 0   100.00%          12                 2    83.33%
```

The `llvm-cov` tool supports specifying a custom demangler, writing out reports in a directory structure, and generating html reports. For the full list of options, please refer to the command guide.

A few final notes:

- The `-sparse` flag is optional but can result in dramatically smaller indexed profiles. This option should not be used if the indexed profile will be reused for PGO.
- Raw profiles can be discarded after they are indexed. Advanced use of the profile runtime library allows an instrumented program to merge profiling information directly into an existing raw profile on disk. The details are out of scope.
- The `llvm-profdata` tool can be used to merge together multiple raw or indexed profiles. To combine profiling data from multiple runs of a program, try e.g:

```
% llvm-profdata merge -sparse foo1.profraw foo2.profdata -o foo3.profdata
```

Exporting coverage data

Coverage data can be exported into JSON using the `llvm-cov export` sub-command. There is a comprehensive reference which defines the structure of the exported data at a high level in the llvm-cov source code.

Interpreting reports

There are five statistics tracked in a coverage summary:

- Function coverage is the percentage of functions which have been executed at least once. A function is considered to be executed if any of its instantiations are executed.
- Instantiation coverage is the percentage of function instantiations which have been executed at least once. Template functions and static inline functions from headers are two kinds of functions which may have multiple instantiations. This statistic is hidden by default in reports, but can be enabled via the `-show-instantiation-summary` option.
- Line coverage is the percentage of code lines which have been executed at least once. Only executable lines within function bodies are considered to be code lines.
- Region coverage is the percentage of code regions which have been executed at least once. A code region may span multiple lines (e.g in a large function body with no control flow). However, it's also possible for a single line to contain multiple code regions (e.g in "return x || y && z").
- Branch coverage is the percentage of "true" and "false" branches that have been taken at least once. Each branch is tied to individual conditions in the source code that may each evaluate to either "true" or "false". These conditions may comprise larger boolean expressions linked by boolean logical operators. For example, "x = (y == 2) || (z < 10)" is a boolean expression that is comprised of two individual conditions, each of which evaluates to either true or false, producing four total branch outcomes.

Of these five statistics, function coverage is usually the least granular while branch coverage is the most granular. 100% branch coverage for a function implies 100% region coverage for a function. The project-wide totals for each statistic are listed in the summary.

Format compatibility guarantees

- There are no backwards or forwards compatibility guarantees for the raw profile format. Raw profiles may be dependent on the specific compiler revision used to generate them. It's inadvisable to store raw profiles for long periods of time.
- Tools must retain **backwards** compatibility with indexed profile formats. These formats are not forwards-compatible: i.e, a tool which uses format version X will not be able to understand format version (X+k).
- Tools must also retain **backwards** compatibility with the format of the coverage mappings emitted into instrumented binaries. These formats are not forwards-compatible.
- The JSON coverage export format has a (major, minor, patch) version triple. Only a major version increment indicates a backwards-incompatible change. A minor version increment is for added functionality, and patch version increments are for bugfixes.

Impact of llvm optimizations on coverage reports

llvm optimizations (such as inlining or CFG simplification) should have no impact on coverage report quality. This is due to the fact that the mapping from source regions to profile counters is immutable, and is generated before the llvm optimizer kicks in. The optimizer can't prove that profile counter instrumentation is safe to delete (because it's not: it affects the profile the program emits), and so leaves it alone.

Note that this coverage feature does not rely on information that can degrade during the course of optimization, such as debug info line tables.

Using the profiling runtime without static initializers

By default the compiler runtime uses a static initializer to determine the profile output path and to register a writer function. To collect profiles without using static initializers, do this manually:

- Export a `int __llvm_profile_runtime` symbol from each instrumented shared library and executable. When the linker finds a definition of this symbol, it knows to skip loading the object which contains the profiling runtime's static initializer.
- Forward-declare `void __llvm_profile_initialize_file(void)` and call it once from each instrumented executable. This function parses `LLVM_PROFILE_FILE`, sets the output path, and truncates any existing files at that path. To get the same behavior without truncating existing files, pass a filename pattern string to `void __llvm_profile_set_filename(char *)`. These calls can be placed anywhere so long as they precede all calls to `__llvm_profile_write_file`.
- Forward-declare `int __llvm_profile_write_file(void)` and call it to write out a profile. This function returns 0 when it succeeds, and a non-zero value otherwise. Calling this function multiple times appends profile data to an existing on-disk raw profile.

In C++ files, declare these as `extern "C"`.

Using the profiling runtime without a filesystem

The profiling runtime also supports freestanding environments that lack a filesystem. The runtime ships as a static archive that's structured to make dependencies on a hosted environment optional, depending on what features the client application uses.

The first step is to export `__llvm_profile_runtime`, as above, to disable the default static initializers. Instead of calling the `*_file()` APIs described above, use the following to save the profile directly to a buffer under your control:

- Forward-declare `uint64_t __llvm_profile_get_size_for_buffer(void)` and call it to determine the size of the profile. You'll need to allocate a buffer of this size.
- Forward-declare `int __llvm_profile_write_buffer(char *Buffer)` and call it to copy the current counters to `Buffer`, which is expected to already be allocated and big enough for the profile.
- Optionally, forward-declare `void __llvm_profile_reset_counters(void)` and call it to reset the counters before entering a specific section to be profiled. This is only useful if there is some setup that should be excluded from the profile.

In C++ files, declare these as `extern "C"`.

Collecting coverage reports for the llvm project

To prepare a coverage report for llvm (and any of its sub-projects), add `-DLLVM_BUILD_INSTRUMENTED_COVERAGE=On` to the cmake configuration. Raw profiles will be written to `$BUILD_DIR/profiles/`. To prepare an html report, run `llvm/utils/prepare-code-coverage-artifact.py`.

To specify an alternate directory for raw profiles, use `-DLLVM_PROFILE_DATA_DIR`. To change the size of the profile merge pool, use `-DLLVM_PROFILE_MERGE_POOL_SIZE`.

Drawbacks and limitations

- Prior to version 2.26, the GNU binutils BFD linker is not able link programs compiled with `-fcoverage-mapping` in its `--gc-sections` mode. Possible workarounds include disabling `--gc-sections`, upgrading to a newer version of BFD, or using the Gold linker.
- Code coverage does not handle unpredictable changes in control flow or stack unwinding in the presence of exceptions precisely. Consider the following function:

```
int f() {
  may_throw();
  return 0;
}
```

  If the call to `may_throw()` propagates an exception into `f`, the code coverage tool may mark the `return` statement as executed even though it is not. A call to `longjmp()` can have similar effects.

Clang implementation details

This section may be of interest to those wishing to understand or improve the clang code coverage implementation.

Gap regions

Gap regions are source regions with counts. A reporting tool cannot set a line execution count to the count from a gap region unless that region is the only one on a line.

Gap regions are used to eliminate unnatural artifacts in coverage reports, such as red “unexecuted” highlights present at the end of an otherwise covered line, or blue “executed” highlights present at the start of a line that is otherwise not executed.

Branch regions

When viewing branch coverage details in source-based file-level sub-views using `--show-branches`, it is recommended that users show all macro expansions (using option `--show-expansions`) since macros may contain hidden branch conditions. The coverage summary report will always include these macro-based boolean expressions in the overall branch coverage count for a function or source file.

Branch coverage is not tracked for constant folded branch conditions since branches are not generated for these cases. In the source-based file-level sub-view, these branches will simply be shown as `[Folded - Ignored]` so that users are informed about what happened.

Branch coverage is tied directly to branch-generating conditions in the source code. Users should not see hidden branches that aren’t actually tied to the source code.

Switch statements

The region mapping for a switch body consists of a gap region that covers the entire body (starting from the ‘{’ in ‘switch (…) {’, and terminating where the last case ends). This gap region has a zero count: this causes “gap” areas in between case statements, which contain no executable code, to appear uncovered.

When a switch case is visited, the parent region is extended: if the parent region has no start location, its start location becomes the start of the case. This is used to support switch statements without a `CompoundStmt` body, in which the switch body and the single case share a count.

For switches with `CompoundStmt` bodies, a new region is created at the start of each switch case.

Branch regions are also generated for each switch case, including the default case. If there is no explicitly defined default case in the source code, a branch region is generated to correspond to the implicit default case that is generated by the compiler. The implicit branch region is tied to the line and column number of the switch statement condition since no source code for the implicit case exists.

Modules

Introduction 702
- Problems with the current model 703
- Semantic import 703
- Problems modules do not solve 704

Using Modules 704
- Objective-C Import declaration 704
- Includes as imports 704
- Module maps 705
- Compilation model 705
- Command-line parameters 705
  - -cc1 Options 707
- Using Prebuilt Modules 707

Module Semantics 708
- Macros 709

Module Map Language 709
- Lexical structure 710
- Module map file 710
- Module declaration 710
  - Requires declaration 711
  - Header declaration 713
  - Umbrella directory declaration 714
  - Submodule declaration 714
  - Export declaration 715
  - Re-export Declaration 716
  - Use declaration 716
  - Link declaration 717
  - Configuration macros declaration 717
  - Conflict declarations 718
- Attributes 718
- Private Module Map Files 718

Modularizing a Platform 719

Future Directions 720

Where To Learn More About Modules 720

Introduction

Most software is built using a number of software libraries, including libraries supplied by the platform, internal libraries built as part of the software itself to provide structure, and third-party libraries. For each library, one needs to access both its interface (API) and its implementation. In the C family of languages, the interface to a library is accessed by including the appropriate header files(s):

```
#include <SomeLib.h>
```

The implementation is handled separately by linking against the appropriate library. For example, by passing `-lSomeLib` to the linker.

Modules provide an alternative, simpler way to use software libraries that provides better compile-time scalability and eliminates many of the problems inherent to using the C preprocessor to access the API of a library.

Problems with the current model

The `#include` mechanism provided by the C preprocessor is a very poor way to access the API of a library, for a number of reasons:

- **Compile-time scalability**: Each time a header is included, the compiler must preprocess and parse the text in that header and every header it includes, transitively. This process must be repeated for every translation unit in the application, which involves a huge amount of redundant work. In a project with *N* translation units and *M* headers included in each translation unit, the compiler is performing *M x N* work even though most of the *M* headers are shared among multiple translation units. C++ is particularly bad, because the compilation model for templates forces a huge amount of code into headers.
- **Fragility**: `#include` directives are treated as textual inclusion by the preprocessor, and are therefore subject to any active macro definitions at the time of inclusion. If any of the active macro definitions happens to collide with a name in the library, it can break the library API or cause compilation failures in the library header itself. For an extreme example, `#define std "The C++ Standard"` and then include a standard library header: the result is a horrific cascade of failures in the C++ Standard Library's implementation. More subtle real-world problems occur when the headers for two different libraries interact due to macro collisions, and users are forced to reorder `#include` directives or introduce `#undef` directives to break the (unintended) dependency.
- **Conventional workarounds**: C programmers have adopted a number of conventions to work around the fragility of the C preprocessor model. Include guards, for example, are required for the vast majority of headers to ensure that multiple inclusion doesn't break the compile. Macro names are written with `LONG_PREFIXED_UPPERCASE_IDENTIFIERS` to avoid collisions, and some library/framework developers even use `__underscored` names in headers to avoid collisions with "normal" names that (by convention) shouldn't even be macros. These conventions are a barrier to entry for developers coming from non-C languages, are boilerplate for more experienced developers, and make our headers far uglier than they should be.
- **Tool confusion**: In a C-based language, it is hard to build tools that work well with software libraries, because the boundaries of the libraries are not clear. Which headers belong to a particular library, and in what order should those headers be included to guarantee that they compile correctly? Are the headers C, C++, Objective-C++, or one of the variants of these languages? What declarations in those headers are actually meant to be part of the API, and what declarations are present only because they had to be written as part of the header file?

Semantic import

Modules improve access to the API of software libraries by replacing the textual preprocessor inclusion model with a more robust, more efficient semantic model. From the user's perspective, the code looks only slightly different, because one uses an `import` declaration rather than a `#include` preprocessor directive:

```
import std.io; // pseudo-code; see below for syntax discussion
```

However, this module import behaves quite differently from the corresponding `#include <stdio.h>`: when the compiler sees the module import above, it loads a binary representation of the `std.io` module and makes its API available to the application directly. Preprocessor definitions that precede the import declaration have no impact on the API provided by `std.io`, because the module itself was compiled as a separate, standalone module. Additionally, any linker flags required to use the `std.io` module will automatically be provided when the module is imported [5] This semantic import model addresses many of the problems of the preprocessor inclusion model:

- **Compile-time scalability**: The `std.io` module is only compiled once, and importing the module into a translation unit is a constant-time operation (independent of module system). Thus, the API of each software library is only parsed once, reducing the *M x N* compilation problem to an *M + N* problem.
- **Fragility**: Each module is parsed as a standalone entity, so it has a consistent preprocessor environment. This completely eliminates the need for `__underscored` names and similarly defensive tricks. Moreover, the current preprocessor definitions when an import declaration is encountered are ignored, so one software library can not affect how another software library is compiled, eliminating include-order dependencies.
- **Tool confusion**: Modules describe the API of software libraries, and tools can reason about and present a module as a representation of that API. Because modules can only be built standalone, tools can rely on the module definition to ensure that they get the complete API for the library. Moreover, modules can specify which languages they work with, so, e.g., one can not accidentally attempt to load a C++ module into a C program.

Problems modules do not solve

Many programming languages have a module or package system, and because of the variety of features provided by these languages it is important to define what modules do *not* do. In particular, all of the following are considered out-of-scope for modules:

- **Rewrite the world's code**: It is not realistic to require applications or software libraries to make drastic or non-backward-compatible changes, nor is it feasible to completely eliminate headers. Modules must interoperate with existing software libraries and allow a gradual transition.
- **Versioning**: Modules have no notion of version information. Programmers must still rely on the existing versioning mechanisms of the underlying language (if any exist) to version software libraries.
- **Namespaces**: Unlike in some languages, modules do not imply any notion of namespaces. Thus, a struct declared in one module will still conflict with a struct of the same name declared in a different module, just as they would if declared in two different headers. This aspect is important for backward compatibility, because (for example) the mangled names of entities in software libraries must not change when introducing modules.
- **Binary distribution of modules**: Headers (particularly C++ headers) expose the full complexity of the language. Maintaining a stable binary module format across architectures, compiler versions, and compiler vendors is technically infeasible.

Using Modules

To enable modules, pass the command-line flag `-fmodules`. This will make any modules-enabled software libraries available as modules as well as introducing any modules-specific syntax. Additional command-line parameters are described in a separate section later.

Objective-C Import declaration

Objective-C provides syntax for importing a module via an *@import declaration*, which imports the named module:

```
@import std;
```

The `@import` declaration above imports the entire contents of the `std` module (which would contain, e.g., the entire C or C++ standard library) and make its API available within the current translation unit. To import only part of a module, one may use dot syntax to specific a particular submodule, e.g.,

```
@import std.io;
```

Redundant import declarations are ignored, and one is free to import modules at any point within the translation unit, so long as the import declaration is at global scope.

At present, there is no C or C++ syntax for import declarations. Clang will track the modules proposal in the C++ committee. See the section Includes as imports to see how modules get imported today.

Includes as imports

The primary user-level feature of modules is the import operation, which provides access to the API of software libraries. However, today's programs make extensive use of `#include`, and it is unrealistic to assume that all of this code will change overnight. Instead, modules automatically translate `#include` directives into the corresponding module import. For example, the include directive

```
#include <stdio.h>
```

will be automatically mapped to an import of the module `std.io`. Even with specific `import` syntax in the language, this particular feature is important for both adoption and backward compatibility: automatic translation of `#include` to `import` allows an application to get the benefits of modules (for all modules-enabled libraries) without any changes to the application itself. Thus, users can easily use modules with one compiler while falling back to the preprocessor-inclusion mechanism with other compilers.

Note

The automatic mapping of `#include` to `import` also solves an implementation problem: importing a module with a definition of some entity (say, a `struct Point`) and then parsing a header containing another definition

> of `struct Point` would cause a redefinition error, even if it is the same `struct Point`. By mapping `#include` to `import`, the compiler can guarantee that it always sees just the already-parsed definition from the module.

While building a module, `#include_next` is also supported, with one caveat. The usual behavior of `#include_next` is to search for the specified filename in the list of include paths, starting from the path *after* the one in which the current file was found. Because files listed in module maps are not found through include paths, a different strategy is used for `#include_next` directives in such files: the list of include paths is searched for the specified header name, to find the first include path that would refer to the current file. `#include_next` is interpreted as if the current file had been found in that path. If this search finds a file named by a module map, the `#include_next` directive is translated into an import, just like for a `#include` directive.``

Module maps

The crucial link between modules and headers is described by a *module map*, which describes how a collection of existing headers maps on to the (logical) structure of a module. For example, one could imagine a module `std` covering the C standard library. Each of the C standard library headers (`<stdio.h>`, `<stdlib.h>`, `<math.h>`, etc.) would contribute to the `std` module, by placing their respective APIs into the corresponding submodule (`std.io`, `std.lib`, `std.math`, etc.). Having a list of the headers that are part of the `std` module allows the compiler to build the `std` module as a standalone entity, and having the mapping from header names to (sub)modules allows the automatic translation of `#include` directives to module imports.

Module maps are specified as separate files (each named `module.modulemap`) alongside the headers they describe, which allows them to be added to existing software libraries without having to change the library headers themselves (in most cases [6]). The actual Module map language is described in a later section.

Note

To actually see any benefits from modules, one first has to introduce module maps for the underlying C standard library and the libraries and headers on which it depends. The section Modularizing a Platform describes the steps one must take to write these module maps.

One can use module maps without modules to check the integrity of the use of header files. To do this, use the `-fimplicit-module-maps` option instead of the `-fmodules` option, or use `-fmodule-map-file=` option to explicitly specify the module map files to load.

Compilation model

The binary representation of modules is automatically generated by the compiler on an as-needed basis. When a module is imported (e.g., by an `#include` of one of the module's headers), the compiler will spawn a second instance of itself [7], with a fresh preprocessing context [8], to parse just the headers in that module. The resulting Abstract Syntax Tree (AST) is then persisted into the binary representation of the module that is then loaded into translation unit where the module import was encountered.

The binary representation of modules is persisted in the *module cache*. Imports of a module will first query the module cache and, if a binary representation of the required module is already available, will load that representation directly. Thus, a module's headers will only be parsed once per language configuration, rather than once per translation unit that uses the module.

Modules maintain references to each of the headers that were part of the module build. If any of those headers changes, or if any of the modules on which a module depends change, then the module will be (automatically) recompiled. The process should never require any user intervention.

Command-line parameters

`-fmodules`

Enable the modules feature.

`-fbuiltin-module-map`

Load the Clang builtins module map file. (Equivalent to `-fmodule-map-file=<resource dir>/include/module.modulemap`)

`-fimplicit-module-maps`

Enable implicit search for module map files named `module.modulemap` and similar. This option is implied by `-fmodules`. If this is disabled with `-fno-implicit-module-maps`, module map files will only be loaded if they are explicitly specified via `-fmodule-map-file` or transitively used by another module map file.

`-fmodules-cache-path=<directory>`

Specify the path to the modules cache. If not provided, Clang will select a system-appropriate default.

`-fno-autolink`

Disable automatic linking against the libraries associated with imported modules.

`-fmodules-ignore-macro=macroname`

Instruct modules to ignore the named macro when selecting an appropriate module variant. Use this for macros defined on the command line that don't affect how modules are built, to improve sharing of compiled module files.

`-fmodules-prune-interval=seconds`

Specify the minimum delay (in seconds) between attempts to prune the module cache. Module cache pruning attempts to clear out old, unused module files so that the module cache itself does not grow without bound. The default delay is large (604,800 seconds, or 7 days) because this is an expensive operation. Set this value to 0 to turn off pruning.

`-fmodules-prune-after=seconds`

Specify the minimum time (in seconds) for which a file in the module cache must be unused (according to access time) before module pruning will remove it. The default delay is large (2,678,400 seconds, or 31 days) to avoid excessive module rebuilding.

`-module-file-info <module file name>`

Debugging aid that prints information about a given module file (with a `.pcm` extension), including the language and preprocessor options that particular module variant was built with.

`-fmodules-decluse`

Enable checking of module `use` declarations.

`-fmodule-name=module-id`

Consider a source file as a part of the given module.

`-fmodule-map-file=<file>`

Load the given module map file if a header from its directory or one of its subdirectories is loaded.

`-fmodules-search-all`

If a symbol is not found, search modules referenced in the current module maps but not imported for symbols, so the error message can reference the module by name. Note that if the global module index has not been built before, this might take some time as it needs to build all the modules. Note that this option doesn't apply in module builds, to avoid the recursion.

`-fno-implicit-modules`

All modules used by the build must be specified with `-fmodule-file`.

`-fmodule-file=[<name>=]<file>`

Specify the mapping of module names to precompiled module files. If the name is omitted, then the module file is loaded whether actually required or not. If the name is specified, then the mapping is treated as another prebuilt module search mechanism (in addition to `-fprebuilt-module-path`) and the module is only loaded if required. Note that in this case the specified file also overrides this module's paths that might be embedded in other precompiled module files.

`-fprebuilt-module-path=<directory>`

Specify the path to the prebuilt modules. If specified, we will look for modules in this directory for a given top-level module name. We don't need a module map for loading prebuilt modules in this directory and the compiler will not try to rebuild these modules. This can be specified multiple times.

`-fprebuilt-implicit-modules`

Enable prebuilt implicit modules. If a prebuilt module is not found in the prebuilt modules paths (specified via `-fprebuilt-module-path`), we will look for a matching implicit module in the prebuilt modules paths.

-cc1 Options

`-fmodules-strict-context-hash`

Enables hashing of all compiler options that could impact the semantics of a module in an implicit build. This includes things such as header search paths and diagnostics. Using this option may lead to an excessive number of modules being built if the command line arguments are not homogeneous across your build.

Using Prebuilt Modules

Below are a few examples illustrating uses of prebuilt modules via the different options.

First, let's set up files for our examples.

```
/* A.h */
#ifdef ENABLE_A
void a() {}
#endif

/* B.h */
#include "A.h"

/* use.c */
#include "B.h"
void use() {
#ifdef ENABLE_A
  a();
#endif
}

/* module.modulemap */
module A {
  header "A.h"
}
module B {
  header "B.h"
  export *
}
```

In the examples below, the compilation of `use.c` can be done without `-cc1`, but the commands used to prebuild the modules would need to be updated to take into account the default options passed to `clang -cc1`. (See `clang use.c -v`) Note also that, since we use `-cc1`, we specify the `-fmodule-map-file=` or `-fimplicit-module-maps` options explicitly. When using the clang driver, `-fimplicit-module-maps` is implied by `-fmodules`.

First let us use an explicit mapping from modules to files.

```
rm -rf prebuilt ; mkdir prebuilt
clang -cc1 -emit-module -o prebuilt/A.pcm -fmodules module.modulemap -fmodule-name=A
clang -cc1 -emit-module -o prebuilt/B.pcm -fmodules module.modulemap -fmodule-name=B -fmodule-file=A=prebuilt/A.pcm
clang -cc1 -emit-obj use.c -fmodules -fmodule-map-file=module.modulemap -fmodule-file=A=prebuilt/A.pcm -fmodule-file=B=prebuilt/B.pcm
```

Instead of of specifying the mappings manually, it can be convenient to use the `-fprebuilt-module-path` option. Let's also use `-fimplicit-module-maps` instead of manually pointing to our module map.

```
rm -rf prebuilt; mkdir prebuilt
clang -cc1 -emit-module -o prebuilt/A.pcm -fmodules module.modulemap -fmodule-name=A
clang -cc1 -emit-module -o prebuilt/B.pcm -fmodules module.modulemap -fmodule-name=B -fprebuilt-module-path=prebuilt
clang -cc1 -emit-obj use.c -fmodules -fimplicit-module-maps -fprebuilt-module-path=prebuilt
```

A trick to prebuild all modules required for our source file in one command is to generate implicit modules while using the `-fdisable-module-hash` option.

```
rm -rf prebuilt ; mkdir prebuilt
clang -cc1 -emit-obj use.c -fmodules -fimplicit-module-maps -fmodules-cache-path=prebuilt -fdisable-module-hash
ls prebuilt/*.pcm
# prebuilt/A.pcm  prebuilt/B.pcm
```

Note that with explicit or prebuilt modules, we are responsible for, and should be particularly careful about the compatibility of our modules. Using mismatching compilation options and modules may lead to issues.

```
clang -cc1 -emit-obj use.c -fmodules -fimplicit-module-maps -fprebuilt-module-path=prebuilt -DENABLE_A
# use.c:4:10: warning: implicit declaration of function 'a' is invalid in C99 [-Wimplicit-function-declaration]
#   return a(x);
#          ^
# 1 warning generated.
```

So we need to maintain multiple versions of prebuilt modules. We can do so using a manual module mapping, or pointing to a different prebuilt module cache path. For example:

```
rm -rf prebuilt ; mkdir prebuilt ; rm -rf prebuilt_a ; mkdir prebuilt_a
clang -cc1 -emit-obj use.c -fmodules -fimplicit-module-maps -fmodules-cache-path=prebuilt -fdisable-module-hash
clang -cc1 -emit-obj use.c -fmodules -fimplicit-module-maps -fmodules-cache-path=prebuilt_a -fdisable-module-hash -DENABLE_A
clang -cc1 -emit-obj use.c -fmodules -fimplicit-module-maps -fprebuilt-module-path=prebuilt
clang -cc1 -emit-obj use.c -fmodules -fimplicit-module-maps -fprebuilt-module-path=prebuilt_a -DENABLE_A
```

Instead of managing the different module versions manually, we can build implicit modules in a given cache path (using `-fmodules-cache-path`), and reuse them as prebuilt implicit modules by passing `-fprebuilt-module-path` and `-fprebuilt-implicit-modules`.

```
rm -rf prebuilt; mkdir prebuilt
clang -cc1 -emit-obj -o use.o use.c -fmodules -fimplicit-module-maps -fmodules-cache-path=prebuilt
clang -cc1 -emit-obj -o use.o use.c -fmodules -fimplicit-module-maps -fmodules-cache-path=prebuilt -DENABLE_A
find prebuilt -name "*.pcm"
# prebuilt/1AYBIGPM8R2GA/A-3L1K4LUA6O31.pcm
# prebuilt/1AYBIGPM8R2GA/B-3L1K4LUA6O31.pcm
# prebuilt/VH0YZMF1OIRK/A-3L1K4LUA6O31.pcm
# prebuilt/VH0YZMF1OIRK/B-3L1K4LUA6O31.pcm
clang -cc1 -emit-obj -o use.o use.c -fmodules -fimplicit-module-maps -fprebuilt-module-path=prebuilt -fprebuilt-implicit-modules
clang -cc1 -emit-obj -o use.o use.c -fmodules -fimplicit-module-maps -fprebuilt-module-path=prebuilt -fprebuilt-implicit-modules -DENABLE_A
```

Finally we want to allow implicit modules for configurations that were not prebuilt. When using the clang driver a module cache path is implicitly selected. Using `-cc1`, we simply add use the `-fmodules-cache-path` option.

```
clang -cc1 -emit-obj -o use.o use.c -fmodules -fimplicit-module-maps -fprebuilt-module-path=prebuilt -fprebuilt-implicit-modules -fmodules-cache-path=cache
clang -cc1 -emit-obj -o use.o use.c -fmodules -fimplicit-module-maps -fprebuilt-module-path=prebuilt -fprebuilt-implicit-modules -fmodules-cache-path=cache -DENABLE_A
clang -cc1 -emit-obj -o use.o use.c -fmodules -fimplicit-module-maps -fprebuilt-module-path=prebuilt -fprebuilt-implicit-modules -fmodules-cache-path=cache -DENABLE_A -DOTHER_OPTIONS
```

This way, a single directory containing multiple variants of modules can be prepared and reused. The options configuring the module cache are independent of other options.

Module Semantics

Modules are modeled as if each submodule were a separate translation unit, and a module import makes names from the other translation unit visible. Each submodule starts with a new preprocessor state and an empty translation unit.

Note

This behavior is currently only approximated when building a module with submodules. Entities within a submodule that has already been built are visible when building later submodules in that module. This can lead to fragile modules that depend on the build order used for the submodules of the module, and should not be relied upon. This behavior is subject to change.

As an example, in C, this implies that if two structs are defined in different submodules with the same name, those two types are distinct types (but may be *compatible* types if their definitions match). In C++, two structs defined with the same name in different submodules are the *same* type, and must be equivalent under C++'s One Definition Rule.

Note

Clang currently only performs minimal checking for violations of the One Definition Rule.

If any submodule of a module is imported into any part of a program, the entire top-level module is considered to be part of the program. As a consequence of this, Clang may diagnose conflicts between an entity declared in an unimported submodule and an entity declared in the current translation unit, and Clang may inline or devirtualize based on knowledge from unimported submodules.

Macros

The C and C++ preprocessor assumes that the input text is a single linear buffer, but with modules this is not the case. It is possible to import two modules that have conflicting definitions for a macro (or where one `#defines` a macro and the other `#undefines` it). The rules for handling macro definitions in the presence of modules are as follows:

- Each definition and undefinition of a macro is considered to be a distinct entity.
- Such entities are *visible* if they are from the current submodule or translation unit, or if they were exported from a submodule that has been imported.
- A `#define X` or `#undef X` directive *overrides* all definitions of `X` that are visible at the point of the directive.
- A `#define` or `#undef` directive is *active* if it is visible and no visible directive overrides it.
- A set of macro directives is *consistent* if it consists of only `#undef` directives, or if all `#define` directives in the set define the macro name to the same sequence of tokens (following the usual rules for macro redefinitions).
- If a macro name is used and the set of active directives is not consistent, the program is ill-formed. Otherwise, the (unique) meaning of the macro name is used.

For example, suppose:

- `<stdio.h>` defines a macro `getc` (and exports its `#define`)
- `<cstdio>` imports the `<stdio.h>` module and undefines the macro (and exports its `#undef`)

The `#undef` overrides the `#define`, and a source file that imports both modules *in any order* will not see `getc` defined as a macro.

Module Map Language

Warning

The module map language is not currently guaranteed to be stable between major revisions of Clang.

The module map language describes the mapping from header files to the logical structure of modules. To enable support for using a library as a module, one must write a `module.modulemap` file for that library. The `module.modulemap` file is placed alongside the header files themselves, and is written in the module map language described below.

Note

For compatibility with previous releases, if a module map file named `module.modulemap` is not found, Clang will also search for a file named `module.map`. This behavior is deprecated and we plan to eventually remove it.

As an example, the module map file for the C standard library might look a bit like this:

```
module std [system] [extern_c] {
  module assert {
    textual header "assert.h"
    header "bits/assert-decls.h"
    export *
  }

  module complex {
    header "complex.h"
    export *
  }

  module ctype {
```

```
    header "ctype.h"
    export *
  }

  module errno {
    header "errno.h"
    header "sys/errno.h"
    export *
  }

  module fenv {
    header "fenv.h"
    export *
  }

  // ...more headers follow...
}
```

Here, the top-level module `std` encompasses the whole C standard library. It has a number of submodules containing different parts of the standard library: `complex` for complex numbers, `ctype` for character types, etc. Each submodule lists one of more headers that provide the contents for that submodule. Finally, the `export *` command specifies that anything included by that submodule will be automatically re-exported.

Lexical structure

Module map files use a simplified form of the C99 lexer, with the same rules for identifiers, tokens, string literals, `/* */` and `//` comments. The module map language has the following reserved words; all other C identifiers are valid identifiers.

```
config_macros export_as  private
conflict      framework  requires
exclude       header     textual
explicit      link       umbrella
extern        module     use
export
```

Module map file

A module map file consists of a series of module declarations:

```
module-map-file:
  module-declaration*
```

Within a module map file, modules are referred to by a *module-id*, which uses periods to separate each part of a module's name:

```
module-id:
  identifier ('.' identifier)*
```

Module declaration

A module declaration describes a module, including the headers that contribute to that module, its submodules, and other aspects of the module.

```
module-declaration:
  explicitopt frameworkopt module module-id attributesopt '{' module-member* '}'
  extern module module-id string-literal
```

The *module-id* should consist of only a single *identifier*, which provides the name of the module being defined. Each module shall have a single definition.

The `explicit` qualifier can only be applied to a submodule, i.e., a module that is nested within another module. The contents of explicit submodules are only made available when the submodule itself was explicitly named in an import declaration or was re-exported from an imported module.

The `framework` qualifier specifies that this module corresponds to a Darwin-style framework. A Darwin-style framework (used primarily on macOS and iOS) is contained entirely in directory `Name.framework`, where `Name` is the name of the framework (and, therefore, the name of the module). That directory has the following layout:

```
Name.framework/
  Modules/module.modulemap  Module map for the framework
  Headers/                  Subdirectory containing framework headers
  PrivateHeaders/           Subdirectory containing framework private headers
  Frameworks/               Subdirectory containing embedded frameworks
  Resources/                Subdirectory containing additional resources
  Name                      Symbolic link to the shared library for the framework
```

The `system` attribute specifies that the module is a system module. When a system module is rebuilt, all of the module's headers will be considered system headers, which suppresses warnings. This is equivalent to placing `#pragma GCC system_header` in each of the module's headers. The form of attributes is described in the section Attributes, below.

The `extern_c` attribute specifies that the module contains C code that can be used from within C++. When such a module is built for use in C++ code, all of the module's headers will be treated as if they were contained within an implicit `extern "C"` block. An import for a module with this attribute can appear within an `extern "C"` block. No other restrictions are lifted, however: the module currently cannot be imported within an `extern "C"` block in a namespace.

The `no_undeclared_includes` attribute specifies that the module can only reach non-modular headers and headers from used modules. Since some headers could be present in more than one search path and map to different modules in each path, this mechanism helps clang to find the right header, i.e., prefer the one for the current module or in a submodule instead of the first usual match in the search paths.

Modules can have a number of different kinds of members, each of which is described below:

```
module-member:
  requires-declaration
  header-declaration
  umbrella-dir-declaration
  submodule-declaration
  export-declaration
  export-as-declaration
  use-declaration
  link-declaration
  config-macros-declaration
  conflict-declaration
```

An extern module references a module defined by the *module-id* in a file given by the *string-literal*. The file can be referenced either by an absolute path or by a path relative to the current map file.

Requires declaration

A *requires-declaration* specifies the requirements that an importing translation unit must satisfy to use the module.

```
requires-declaration:
  requires feature-list

feature-list:
  feature (',' feature)*

feature:
  !opt identifier
```

The requirements clause allows specific modules or submodules to specify that they are only accessible with certain language dialects, platforms, environments and target specific features. The feature list is a set of identifiers, defined below. If any of the features is not available in a given translation unit, that translation unit shall not import the

module. When building a module for use by a compilation, submodules requiring unavailable features are ignored. The optional `!` indicates that a feature is incompatible with the module.

The following features are defined:

altivec
: The target supports AltiVec.

blocks
: The “blocks” language feature is available.

coroutines
: Support for the coroutines TS is available.

cplusplus
: C++ support is available.

cplusplus11
: C++11 support is available.

cplusplus14
: C++14 support is available.

cplusplus17
: C++17 support is available.

c99
: C99 support is available.

c11
: C11 support is available.

c17
: C17 support is available.

freestanding
: A freestanding environment is available.

gnuinlineasm
: GNU inline ASM is available.

objc
: Objective-C support is available.

objc_arc
: Objective-C Automatic Reference Counting (ARC) is available

opencl
: OpenCL is available

tls
: Thread local storage is available.

target feature
: A specific target feature (e.g., `sse4`, `avx`, `neon`) is available.

platform/os
: A os/platform variant (e.g. `freebsd`, `win32`, `windows`, `linux`, `ios`, `macos`, `iossimulator`) is available.

environment
: A environment variant (e.g. `gnu`, `gnueabi`, `android`, `msvc`) is available.

Example: The `std` module can be extended to also include C++ and C++11 headers using a *requires-declaration*:

```
module std {
   // C standard library...

   module vector {
     requires cplusplus
     header "vector"
```

```
    }

    module type_traits {
      requires cplusplus11
      header "type_traits"
    }
  }
```

Header declaration

A header declaration specifies that a particular header is associated with the enclosing module.

```
header-declaration:
  privateopt textualopt header string-literal header-attrsopt
  umbrella header string-literal header-attrsopt
  exclude header string-literal header-attrsopt

header-attrs:
  '{' header-attr* '}'

header-attr:
  size integer-literal
  mtime integer-literal
```

A header declaration that does not contain `exclude` nor `textual` specifies a header that contributes to the enclosing module. Specifically, when the module is built, the named header will be parsed and its declarations will be (logically) placed into the enclosing submodule.

A header with the `umbrella` specifier is called an umbrella header. An umbrella header includes all of the headers within its directory (and any subdirectories), and is typically used (in the `#include` world) to easily access the full API provided by a particular library. With modules, an umbrella header is a convenient shortcut that eliminates the need to write out `header` declarations for every library header. A given directory can only contain a single umbrella header.

Note

Any headers not included by the umbrella header should have explicit `header` declarations. Use the `-Wincomplete-umbrella` warning option to ask Clang to complain about headers not covered by the umbrella header or the module map.

A header with the `private` specifier may not be included from outside the module itself.

A header with the `textual` specifier will not be compiled when the module is built, and will be textually included if it is named by a `#include` directive. However, it is considered to be part of the module for the purpose of checking *use-declaration*s, and must still be a lexically-valid header file. In the future, we intend to pre-tokenize such headers and include the token sequence within the prebuilt module representation.

A header with the `exclude` specifier is excluded from the module. It will not be included when the module is built, nor will it be considered to be part of the module, even if an `umbrella` header or directory would otherwise make it part of the module.

Example: The C header `assert.h` is an excellent candidate for a textual header, because it is meant to be included multiple times (possibly with different `NDEBUG` settings). However, declarations within it should typically be split into a separate modular header.

```
module std [system] {
  textual header "assert.h"
}
```

A given header shall not be referenced by more than one *header-declaration*.

Two *header-declaration*s, or a *header-declaration* and a `#include`, are considered to refer to the same file if the paths resolve to the same file and the specified *header-attr*s (if any) match the attributes of that file, even if the file is named differently (for instance, by a relative path or via symlinks).

Note

The use of *header-attr*s avoids the need for Clang to speculatively `stat` every header referenced by a module map. It is recommended that *header-attr*s only be used in machine-generated module maps, to avoid mismatches between attribute values and the corresponding files.

Umbrella directory declaration

An umbrella directory declaration specifies that all of the headers in the specified directory should be included within the module.

```
umbrella-dir-declaration:
  umbrella string-literal
```

The *string-literal* refers to a directory. When the module is built, all of the header files in that directory (and its subdirectories) are included in the module.

An *umbrella-dir-declaration* shall not refer to the same directory as the location of an umbrella *header-declaration*. In other words, only a single kind of umbrella can be specified for a given directory.

Note

Umbrella directories are useful for libraries that have a large number of headers but do not have an umbrella header.

Submodule declaration

Submodule declarations describe modules that are nested within their enclosing module.

```
submodule-declaration:
  module-declaration
  inferred-submodule-declaration
```

A *submodule-declaration* that is a *module-declaration* is a nested module. If the *module-declaration* has a `framework` specifier, the enclosing module shall have a `framework` specifier; the submodule's contents shall be contained within the subdirectory `Frameworks/SubName.framework`, where `SubName` is the name of the submodule.

A *submodule-declaration* that is an *inferred-submodule-declaration* describes a set of submodules that correspond to any headers that are part of the module but are not explicitly described by a *header-declaration*.

```
inferred-submodule-declaration:
  explicitopt frameworkopt module '*' attributesopt '{' inferred-submodule-member* '}'

inferred-submodule-member:
  export '*'
```

A module containing an *inferred-submodule-declaration* shall have either an umbrella header or an umbrella directory. The headers to which the *inferred-submodule-declaration* applies are exactly those headers included by the umbrella header (transitively) or included in the module because they reside within the umbrella directory (or its subdirectories).

For each header included by the umbrella header or in the umbrella directory that is not named by a *header-declaration*, a module declaration is implicitly generated from the *inferred-submodule-declaration*. The module will:

- Have the same name as the header (without the file extension)

- Have the `explicit` specifier, if the *inferred-submodule-declaration* has the `explicit` specifier
- Have the `framework` specifier, if the *inferred-submodule-declaration* has the `framework` specifier
- Have the attributes specified by the *inferred-submodule-declaration*
- Contain a single *header-declaration* naming that header
- Contain a single *export-declaration* `export *`, if the *inferred-submodule-declaration* contains the *inferred-submodule-member* `export *`

Example: If the subdirectory “MyLib” contains the headers `A.h` and `B.h`, then the following module map:

```
module MyLib {
  umbrella "MyLib"
  explicit module * {
    export *
  }
}
```

is equivalent to the (more verbose) module map:

```
module MyLib {
  explicit module A {
    header "A.h"
    export *
  }

  explicit module B {
    header "B.h"
    export *
  }
}
```

Export declaration

An *export-declaration* specifies which imported modules will automatically be re-exported as part of a given module’s API.

```
export-declaration:
  export wildcard-module-id

wildcard-module-id:
  identifier
  '*'
  identifier '.' wildcard-module-id
```

The *export-declaration* names a module or a set of modules that will be re-exported to any translation unit that imports the enclosing module. Each imported module that matches the *wildcard-module-id* up to, but not including, the first * will be re-exported.

Example: In the following example, importing `MyLib.Derived` also provides the API for `MyLib.Base`:

```
module MyLib {
  module Base {
    header "Base.h"
  }

  module Derived {
    header "Derived.h"
    export Base
  }
}
```

Note that, if `Derived.h` includes `Base.h`, one can simply use a wildcard export to re-export everything `Derived.h` includes:

```
module MyLib {
  module Base {
    header "Base.h"
  }

  module Derived {
    header "Derived.h"
    export *
  }
}
```

Note

The wildcard export syntax `export *` re-exports all of the modules that were imported in the actual header file. Because `#include` directives are automatically mapped to module imports, `export *` provides the same transitive-inclusion behavior provided by the C preprocessor, e.g., importing a given module implicitly imports all of the modules on which it depends. Therefore, liberal use of `export *` provides excellent backward compatibility for programs that rely on transitive inclusion (i.e., all of them).

Re-export Declaration

An *export-as-declaration* specifies that the current module will have its interface re-exported by the named module.

```
export-as-declaration:
  export_as identifier
```

The *export-as-declaration* names the module that the current module will be re-exported through. Only top-level modules can be re-exported, and any given module may only be re-exported through a single module.

Example: In the following example, the module `MyFrameworkCore` will be re-exported via the module `MyFramework`:

```
module MyFrameworkCore {
  export_as MyFramework
}
```

Use declaration

A *use-declaration* specifies another module that the current top-level module intends to use. When the option *-fmodules-decluse* is specified, a module can only use other modules that are explicitly specified in this way.

```
use-declaration:
  use module-id
```

Example: In the following example, use of A from C is not declared, so will trigger a warning.

```
module A {
  header "a.h"
}

module B {
  header "b.h"
}

module C {
  header "c.h"
  use B
}
```

When compiling a source file that implements a module, use the option `-fmodule-name=module-id` to indicate that the source file is logically part of that module.

The compiler at present only applies restrictions to the module directly being built.

Link declaration

A *link-declaration* specifies a library or framework against which a program should be linked if the enclosing module is imported in any translation unit in that program.

```
link-declaration:
  link frameworkopt string-literal
```

The *string-literal* specifies the name of the library or framework against which the program should be linked. For example, specifying “clangBasic” would instruct the linker to link with `-lclangBasic` for a Unix-style linker.

A *link-declaration* with the `framework` specifies that the linker should link against the named framework, e.g., with `-framework MyFramework`.

Note

Automatic linking with the `link` directive is not yet widely implemented, because it requires support from both the object file format and the linker. The notion is similar to Microsoft Visual Studio’s `#pragma comment(lib...)`.

Configuration macros declaration

The *config-macros-declaration* specifies the set of configuration macros that have an effect on the API of the enclosing module.

```
config-macros-declaration:
  config_macros attributesopt config-macro-listopt

config-macro-list:
  identifier (',' identifier)*
```

Each *identifier* in the *config-macro-list* specifies the name of a macro. The compiler is required to maintain different variants of the given module for differing definitions of any of the named macros.

A *config-macros-declaration* shall only be present on a top-level module, i.e., a module that is not nested within an enclosing module.

The `exhaustive` attribute specifies that the list of macros in the *config-macros-declaration* is exhaustive, meaning that no other macro definition is intended to have an effect on the API of that module.

Note

The `exhaustive` attribute implies that any macro definitions for macros not listed as configuration macros should be ignored completely when building the module. As an optimization, the compiler could reduce the number of unique module variants by not considering these non-configuration macros. This optimization is not yet implemented in Clang.

A translation unit shall not import the same module under different definitions of the configuration macros.

Note

Clang implements a weak form of this requirement: the definitions used for configuration macros are fixed based on the definitions provided by the command line. If an import occurs and the definition of any configuration macro has changed, the compiler will produce a warning (under the control of `-Wconfig-macros`).

Example: A logging library might provide different API (e.g., in the form of different definitions for a logging macro) based on the `NDEBUG` macro setting:

```
module MyLogger {
  umbrella header "MyLogger.h"
  configmacros [exhaustive] NDEBUG
}
```

Conflict declarations

A *conflict-declaration* describes a case where the presence of two different modules in the same translation unit is likely to cause a problem. For example, two modules may provide similar-but-incompatible functionality.

```
conflict-declaration:
  conflict module-id ',' string-literal
```

The *module-id* of the *conflict-declaration* specifies the module with which the enclosing module conflicts. The specified module shall not have been imported in the translation unit when the enclosing module is imported.

The *string-literal* provides a message to be provided as part of the compiler diagnostic when two modules conflict.

Note

Clang emits a warning (under the control of `-Wmodule-conflict`) when a module conflict is discovered.

Example:

```
module Conflicts {
  explicit module A {
    header "conflict_a.h"
    conflict B, "we just don't like B"
  }

  module B {
    header "conflict_b.h"
  }
}
```

Attributes

Attributes are used in a number of places in the grammar to describe specific behavior of other declarations. The format of attributes is fairly simple.

```
attributes:
  attribute attributesopt

attribute:
  '[' identifier ']'
```

Any *identifier* can be used as an attribute, and each declaration specifies what attributes can be applied to it.

Private Module Map Files

Module map files are typically named `module.modulemap` and live either alongside the headers they describe or in a parent directory of the headers they describe. These module maps typically describe all of the API for the library.

However, in some cases, the presence or absence of particular headers is used to distinguish between the “public” and “private” APIs of a particular library. For example, a library may contain the headers `Foo.h` and `Foo_Private.h`, providing public and private APIs, respectively. Additionally, `Foo_Private.h` may only be available on some versions of library, and absent in others. One cannot easily express this with a single module map file in the library:

```
module Foo {
  header "Foo.h"
  ...
}

module Foo_Private {
  header "Foo_Private.h"
  ...
}
```

because the header `Foo_Private.h` won't always be available. The module map file could be customized based on whether `Foo_Private.h` is available or not, but doing so requires custom build machinery.

Private module map files, which are named `module.private.modulemap` (or, for backward compatibility, `module_private.map`), allow one to augment the primary module map file with an additional modules. For example, we would split the module map file above into two module map files:

```
/* module.modulemap */
module Foo {
  header "Foo.h"
}

/* module.private.modulemap */
module Foo_Private {
  header "Foo_Private.h"
}
```

When a `module.private.modulemap` file is found alongside a `module.modulemap` file, it is loaded after the `module.modulemap` file. In our example library, the `module.private.modulemap` file would be available when `Foo_Private.h` is available, making it easier to split a library's public and private APIs along header boundaries.

When writing a private module as part of a *framework*, it's recommended that:

- Headers for this module are present in the `PrivateHeaders` framework subdirectory.
- The private module is defined as a *top level module* with the name of the public framework prefixed, like `Foo_Private` above. Clang has extra logic to work with this naming, using `FooPrivate` or `Foo.Private` (submodule) trigger warnings and might not work as expected.

Modularizing a Platform

To get any benefit out of modules, one needs to introduce module maps for software libraries starting at the bottom of the stack. This typically means introducing a module map covering the operating system's headers and the C standard library headers (in `/usr/include`, for a Unix system).

The module maps will be written using the module map language, which provides the tools necessary to describe the mapping between headers and modules. Because the set of headers differs from one system to the next, the module map will likely have to be somewhat customized for, e.g., a particular distribution and version of the operating system. Moreover, the system headers themselves may require some modification, if they exhibit any anti-patterns that break modules. Such common patterns are described below.

Macro-guarded copy-and-pasted definitions

System headers vend core types such as `size_t` for users. These types are often needed in a number of system headers, and are almost trivial to write. Hence, it is fairly common to see a definition such as the following copy-and-pasted throughout the headers:

```
#ifndef _SIZE_T
#define _SIZE_T
typedef __SIZE_TYPE__ size_t;
#endif
```

Unfortunately, when modules compiles all of the C library headers together into a single module, only the first actual type definition of `size_t` will be visible, and then only in the submodule corresponding to the lucky first header. Any other headers that have copy-and-pasted versions of this pattern will *not* have a definition of `size_t`. Importing the submodule corresponding to one of those headers will therefore not yield `size_t` as

part of the API, because it wasn't there when the header was parsed. The fix for this problem is either to pull the copied declarations into a common header that gets included everywhere `size_t` is part of the API, or to eliminate the `#ifndef` and redefine the `size_t` type. The latter works for C++ headers and C11, but will cause an error for non-modules C90/C99, where redefinition of `typedefs` is not permitted.

Conflicting definitions

Different system headers may provide conflicting definitions for various macros, functions, or types. These conflicting definitions don't tend to cause problems in a pre-modules world unless someone happens to include both headers in one translation unit. Since the fix is often simply "don't do that", such problems persist. Modules requires that the conflicting definitions be eliminated or that they be placed in separate modules (the former is generally the better answer).

Missing includes

Headers are often missing `#include` directives for headers that they actually depend on. As with the problem of conflicting definitions, this only affects unlucky users who don't happen to include headers in the right order. With modules, the headers of a particular module will be parsed in isolation, so the module may fail to build if there are missing includes.

Headers that vend multiple APIs at different times

Some systems have headers that contain a number of different kinds of API definitions, only some of which are made available with a given include. For example, the header may vend `size_t` only when the macro `__need_size_t` is defined before that header is included, and also vend `wchar_t` only when the macro `__need_wchar_t` is defined. Such headers are often included many times in a single translation unit, and will have no include guards. There is no sane way to map this header to a submodule. One can either eliminate the header (e.g., by splitting it into separate headers, one per actual API) or simply `exclude` it in the module map.

To detect and help address some of these problems, the `clang-tools-extra` repository contains a `modularize` tool that parses a set of given headers and attempts to detect these problems and produce a report. See the tool's in-source documentation for information on how to check your system or library headers.

Future Directions

Modules support is under active development, and there are many opportunities remaining to improve it. Here are a few ideas:

Detect unused module imports

Unlike with `#include` directives, it should be fairly simple to track whether a directly-imported module has ever been used. By doing so, Clang can emit `unused import` or `unused #include` diagnostics, including Fix-Its to remove the useless imports/includes.

Fix-Its for missing imports

It's fairly common for one to make use of some API while writing code, only to get a compiler error about "unknown type" or "no function named" because the corresponding header has not been included. Clang can detect such cases and auto-import the required module, but should provide a Fix-It to add the import.

Improve modularize

The modularize tool is both extremely important (for deployment) and extremely crude. It needs better UI, better detection of problems (especially for C++), and perhaps an assistant mode to help write module maps for you.

Where To Learn More About Modules

The Clang source code provides additional information about modules:

`clang/lib/Headers/module.modulemap`

Module map for Clang's compiler-specific header files.

`clang/test/Modules/`

Tests specifically related to modules functionality.

`clang/include/clang/Basic/Module.h`

The `Module` class in this header describes a module, and is used throughout the compiler to implement modules.

`clang/include/clang/Lex/ModuleMap.h`

The `ModuleMap` class in this header describes the full module map, consisting of all of the module map files that have been parsed, and providing facilities for looking up module maps and mapping between modules and headers (in both directions).

PCHInternals

Information about the serialized AST format used for precompiled headers and modules. The actual implementation is in the `clangSerialization` library.

5 Automatic linking against the libraries of modules requires specific linker support, which is not widely available.

6 There are certain anti-patterns that occur in headers, particularly system headers, that cause problems for modules. The section Modularizing a Platform describes some of them.

7 The second instance is actually a new thread within the current process, not a separate process. However, the original compiler instance is blocked on the execution of this thread.

8 The preprocessing context in which the modules are parsed is actually dependent on the command-line options provided to the compiler, including the language dialect and any `-D` options. However, the compiled modules for different command-line options are kept distinct, and any preprocessor directives that occur within the translation unit are ignored. See the section on the Configuration macros declaration for more information.

MSVC compatibility

When Clang compiles C++ code for Windows, it attempts to be compatible with MSVC. There are multiple dimensions to compatibility.

First, Clang attempts to be ABI-compatible, meaning that Clang-compiled code should be able to link against MSVC-compiled code successfully. However, C++ ABIs are particularly large and complicated, and Clang's support for MSVC's C++ ABI is a work in progress. If you don't require MSVC ABI compatibility or don't want to use Microsoft's C and C++ runtimes, the mingw32 toolchain might be a better fit for your project.

Second, Clang implements many MSVC language extensions, such as `__declspec(dllexport)` and a handful of pragmas. These are typically controlled by `-fms-extensions`.

Third, MSVC accepts some C++ code that Clang will typically diagnose as invalid. When these constructs are present in widely included system headers, Clang attempts to recover and continue compiling the user's program. Most parsing and semantic compatibility tweaks are controlled by `-fms-compatibility` and `-fdelayed-template-parsing`, and they are a work in progress.

Finally, there is clang-cl, a driver program for clang that attempts to be compatible with MSVC's cl.exe.

ABI features

The status of major ABI-impacting C++ features:

- Record layout: Complete. We've tested this with a fuzzer and have fixed all known bugs.
- Class inheritance: Mostly complete. This covers all of the standard OO features you would expect: virtual method inheritance, multiple inheritance, and virtual inheritance. Every so often we uncover a bug where our tables are incompatible, but this is pretty well in hand. This feature has also been fuzz tested.
- Name mangling: Ongoing. Every new C++ feature generally needs its own mangling. For example, member pointer template arguments have an interesting and distinct mangling. Fortunately, incorrect manglings usually do not result in runtime errors. Non-inline functions with incorrect manglings usually result in link errors, which are relatively easy to diagnose. Incorrect manglings for inline functions and templates result in multiple copies in the final image. The C++ standard requires that those addresses be equal, but few programs rely on this.
- Member pointers: Mostly complete. Standard C++ member pointers are fully implemented and should be ABI compatible. Both #pragma pointers_to_members and the /vm flags are supported. However, MSVC supports an extension to allow creating a pointer to a member of a virtual base class. Clang does not yet support this.
- Debug info: Mostly complete. Clang emits relatively complete CodeView debug information if `/Z7` or `/Zi` is passed. Microsoft's link.exe will transform the CodeView debug information into a PDB that works in Windows debuggers and other tools that consume PDB files like ETW. Work to teach lld about CodeView and PDBs is ongoing.

- RTTI: Complete. Generation of RTTI data structures has been finished, along with support for the `/GR` flag.
- C++ Exceptions: Mostly complete. Support for C++ exceptions (`try` / `catch` / `throw`) have been implemented for x86 and x64. Our implementation has been well tested but we still get the odd bug report now and again. C++ exception specifications are ignored, but this is consistent with Visual C++.
- Asynchronous Exceptions (SEH): Partial. Structured exceptions (`__try` / `__except` / `__finally`) mostly work on x86 and x64. LLVM does not model asynchronous exceptions, so it is currently impossible to catch an asynchronous exception generated in the same frame as the catching `__try`.
- Thread-safe initialization of local statics: Complete. MSVC 2015 added support for thread-safe initialization of such variables by taking an ABI break. We are ABI compatible with both the MSVC 2013 and 2015 ABI for static local variables.
- Lambdas: Mostly complete. Clang is compatible with Microsoft's implementation of lambdas except for providing overloads for conversion to function pointer for different calling conventions. However, Microsoft's extension is non-conforming.

Template instantiation and name lookup

MSVC allows many invalid constructs in class templates that Clang has historically rejected. In order to parse widely distributed headers for libraries such as the Active Template Library (ATL) and Windows Runtime Library (WRL), some template rules have been relaxed or extended in Clang on Windows.

The first major semantic difference is that MSVC appears to defer all parsing an analysis of inline method bodies in class templates until instantiation time. By default on Windows, Clang attempts to follow suit. This behavior is controlled by the `-fdelayed-template-parsing` flag. While Clang delays parsing of method bodies, it still parses the bodies *before* template argument substitution, which is not what MSVC does. The following compatibility tweaks are necessary to parse the template in those cases.

MSVC allows some name lookup into dependent base classes. Even on other platforms, this has been a frequently asked question for Clang users. A dependent base class is a base class that depends on the value of a template parameter. Clang cannot see any of the names inside dependent bases while it is parsing your template, so the user is sometimes required to use the `typename` keyword to assist the parser. On Windows, Clang attempts to follow the normal lookup rules, but if lookup fails, it will assume that the user intended to find the name in a dependent base. While parsing the following program, Clang will recover as if the user had written the commented-out code:

```
template <typename T>
struct Foo : T {
  void f() {
    /*typename*/ T::UnknownType x =  /*this->*/unknownMember;
  }
};
```

After recovery, Clang warns the user that this code is non-standard and issues a hint suggesting how to fix the problem.

As of this writing, Clang is able to compile a simple ATL hello world application. There are still issues parsing WRL headers for modern Windows 8 apps, but they should be addressed soon.

Misexpect

Contents

Misexpect **722**

When developers use `llvm.expect` intrinsics, i.e., through use of `__builtin_expect(...)`, they are trying to communicate how their code is expected to behave at runtime to the optimizer. These annotations, however, can be incorrect for a variety of reasons: changes to the code base invalidate them silently, the developer mis-annotated them (e.g., using `LIKELY` instead of `UNLIKELY`), or perhaps they assumed something incorrectly when they wrote the annotation. Regardless of why, it is useful to detect these situations so that the optimizer can make more useful decisions about the code.

MisExpect diagnostics are intended to help developers identify and address these situations, by comparing the branch weights added by the `llvm.expect` intrinsic to those collected through profiling. Whenever these values are mismatched, a diagnostic is surfaced to the user. Details on how the checks operate in the LLVM backed can be found in LLVM's documentation.

By default MisExpect checking is quite strict, because the use of the `llvm.expect` intrinsic is designed for specialized cases, where the outcome of a condition is severely skewed. As a result, the optimizer can be extremely aggressive, which can result in performance degradation if the outcome is less predictable than the annotation suggests. Even when the annotation is correct 90% of the time, it may be beneficial to either remove the annotation or to use a different intrinsic that can communicate the probability more directly.

Because this may be too strict, MisExpect diagnostics are not enabled by default, and support an additional flag to tolerate some deviation from the exact thresholds. The `-fdiagnostic-misexpect-tolerance=N` accepts deviations when comparing branch weights within `N%` of the expected values. So passing `-fdiagnostic-misexpect-tolerance=5` will not report diagnostic messages if the branch weight from the profile is within 5% of the weight added by the `llvm.expect` intrinsic.

MisExpect diagnostics are also available in the form of optimization remarks, which can be serialized and processed through the `opt-viewer.py` scripts in LLVM.

`-Rpass``=misexpect`
  Enables optimization remarks for misexpect when profiling data conflicts with use of `llvm.expect` intrinsics.

`-Wmisexpect`
  Enables misexpect warnings when profiling data conflicts with use of `llvm.expect` intrinsics.

`-fdiagnostic-misexpect-tolerance``=N`
  Relaxes misexpect checking to tolerate profiling values within N% of the expected branch weight. e.g., a value of `N=5` allows misexpect to check against `0.95 * Threshold`

LLVM supports 4 types of profile formats: Frontend, IR, CS-IR, and Sampling. MisExpect Diagnostics are compatible with all Profiling formats.

Profile Type	Description
Frontend	Profiling instrumentation added during compilation by the frontend, i.e. `clang`
IR	Profiling instrumentation added during by the LLVM backend
CS-IR	Context Sensitive IR based profiles
Sampling	Profiles collected through sampling with external tools, such as `perf` on Linux

OpenCL Support **724**
- Missing features or with limited support 724
- Internals Manual 724
  - OpenCL Metadata 724
  - OpenCL Specific Options 724
  - OpenCL builtins 725
  - OpenCL Extensions and Features 725
    - Implementation guidelines 726
  - Address spaces attribute 726
- C++ for OpenCL Implementation Status 727
  - Missing features or with limited support 727
- OpenCL C 3.0 Usage 727
  - OpenCL C 3.0 Implementation Status 727
- Experimental features 728
  - C++ libraries for OpenCL 728

OpenCL Support

Clang has complete support of OpenCL C versions from 1.0 to 3.0. Support for OpenCL 3.0 is in experimental phase (OpenCL 3.0).

Clang also supports the C++ for OpenCL kernel language.

There are also other new and experimental features available.

Details about usage of clang for OpenCL can be found in Clang Compiler User's Manual.

Missing features or with limited support

- For general issues and bugs with OpenCL in clang refer to the GitHub issue list.
- Command-line flag -cl-ext (used to override extensions/ features supported by a target) is missing support of some functionality i.e. that is implemented fully through libraries (see library-based features and extensions).

Internals Manual

This section acts as internal documentation for OpenCL features design as well as some important implementation aspects. It is primarily targeted at the advanced users and the toolchain developers integrating frontend functionality as a component.

OpenCL Metadata

Clang uses metadata to provide additional OpenCL semantics in IR needed for backends and OpenCL runtime.

Each kernel will have function metadata attached to it, specifying the arguments. Kernel argument metadata is used to provide source level information for querying at runtime, for example using the clGetKernelArgInfo call.

Note that `-cl-kernel-arg-info` enables more information about the original kernel code to be added e.g. kernel parameter names will appear in the OpenCL metadata along with other information.

The IDs used to encode the OpenCL's logical address spaces in the argument info metadata follows the SPIR address space mapping as defined in the SPIR specification section 2.2

OpenCL Specific Options

In addition to the options described in Clang Compiler User's Manual there are the following options specific to the OpenCL frontend.

All the options in this section are frontend-only and therefore if used with regular clang driver they require frontend forwarding, e.g. `-cc1` or `-Xclang`.

`-finclude-default-header`

Adds most of builtin types and function declarations during compilations. By default the OpenCL headers are not loaded by the frontend and therefore certain builtin types and most of builtin functions are not declared. To load them automatically this flag can be passed to the frontend (see also the section on the OpenCL Header):

```
$ clang -Xclang -finclude-default-header test.cl
```

Alternatively the internal header *opencl-c.h* containing the declarations can be included manually using `-include` or `-I` followed by the path to the header location. The header can be found in the clang source tree or installation directory.

```
$ clang -I<path to clang sources>/lib/Headers/opencl-c.h test.cl
$ clang -I<path to clang installation>/lib/clang/<llvm version>/include/opencl-c.h/opencl-c.h test.cl
```

In this example it is assumed that the kernel code contains `#include <opencl-c.h>` just as a regular C include.

Because the header is very large and long to parse, PCH (Precompiled Header and Modules Internals) and modules (Modules) can be used internally to improve the compilation speed.

To enable modules for OpenCL:

```
$ clang -target spir-unknown-unknown -c -emit-llvm -Xclang -finclude-default-header -fmodules -fimplicit-module-maps -fmodules-cache-path=<path to the generated module> test.cl
```

Another way to circumvent long parsing latency for the OpenCL builtin declarations is to use mechanism enabled by -fdeclare-opencl-builtins flag that is available as an alternative feature.

`-fdeclare-opencl-builtins`

In addition to regular header includes with builtin types and functions using -finclude-default-header, clang supports a fast mechanism to declare builtin functions with `-fdeclare-opencl-builtins`. This does not declare the builtin types and therefore it has to be used in combination with `-finclude-default-header` if full functionality is required.

Example of Use:

```
$ clang -Xclang -fdeclare-opencl-builtins test.cl
```

`-ffake-address-space-map`

Overrides the target address space map with a fake map. This allows adding explicit address space IDs to the bitcode for non-segmented memory architectures that do not have separate IDs for each of the OpenCL logical address spaces by default. Passing `-ffake-address-space-map` will add/override address spaces of the target compiled for with the following values: `1-global`, `2-constant`, `3-local`, `4-generic`. The private address space is represented by the absence of an address space attribute in the IR (see also the section on the address space attribute).

```
$ clang -cc1 -ffake-address-space-map test.cl
```

OpenCL builtins

Clang builtins

There are some standard OpenCL functions that are implemented as Clang builtins:

- All pipe functions from section 6.13.16.2/6.13.16.3 of the OpenCL v2.0 kernel language specification.
- Address space qualifier conversion functions `to_global`/`to_local`/`to_private` from section 6.13.9.
- All the `enqueue_kernel` functions from section 6.13.17.1 and enqueue query functions from section 6.13.17.5.

Fast builtin function declarations

The implementation of the fast builtin function declarations (available via the -fdeclare-opencl-builtins option) consists of the following main components:

- A TableGen definitions file `OpenCLBuiltins.td`. This contains a compact representation of the supported builtin functions. When adding new builtin function declarations, this is normally the only file that needs modifying.
- A Clang TableGen emitter defined in `ClangOpenCLBuiltinEmitter.cpp`. During Clang build time, the emitter reads the TableGen definition file and generates `OpenCLBuiltins.inc`. This generated file contains various tables and functions that capture the builtin function data from the TableGen definitions in a compact manner.
- OpenCL specific code in `SemaLookup.cpp`. When `Sema::LookupBuiltin` encounters a potential builtin function, it will check if the name corresponds to a valid OpenCL builtin function. If so, all overloads of the function are inserted using `InsertOCLBuiltinDeclarationsFromTable` and overload resolution takes place.

OpenCL Extensions and Features

Clang implements various extensions to OpenCL kernel languages.

New functionality is accepted as soon as the documentation is detailed to the level sufficient to be implemented. There should be an evidence that the extension is designed with implementation feasibility in consideration and assessment of complexity for C/C++ based compilers. Alternatively, the documentation can be accepted in a format of a draft that can be further refined during the implementation.

Implementation guidelines

This section explains how to extend clang with the new functionality.

Parsing functionality

If an extension modifies the standard parsing it needs to be added to the clang frontend source code. This also means that the associated macro indicating the presence of the extension should be added to clang.

The default flow for adding a new extension into the frontend is to modify OpenCLExtensions.def, containing the list of all extensions and optional features supported by the frontend.

This will add the macro automatically and also add a field in the target options `clang::TargetOptions::OpenCLFeaturesMap` to control the exposure of the new extension during the compilation.

Note that by default targets like *SPIR-V*, *SPIR* or *X86* expose all the OpenCL extensions. For all other targets the configuration has to be made explicitly.

Note that the target extension support performed by clang can be overridden with -cl-ext command-line flags.

Library functionality

If an extension adds functionality that does not modify standard language parsing it should not require modifying anything other than header files and `OpenCLBuiltins.td` detailed in OpenCL builtins. Most commonly such extensions add functionality via libraries (by adding non-native types or functions) parsed regularly. Similar to other languages this is the most common way to add new functionality.

Clang has standard headers where new types and functions are being added, for more details refer to the section on the OpenCL Header. The macros indicating the presence of such extensions can be added in the standard header files conditioned on target specific predefined macros or/and language version predefined macros (see feature/extension preprocessor macros defined in opencl-c-base.h).

Pragmas

Some extensions alter standard parsing dynamically via pragmas.

Clang provides a mechanism to add the standard extension pragma `OPENCL EXTENSION` by setting a dedicated flag in the extension list entry of `OpenCLExtensions.def`. Note that there is no default behavior for the standard extension pragmas as it is not specified (for the standards up to and including version 3.0) in a sufficient level of detail and, therefore, there is no default functionality provided by clang.

Pragmas without detailed information of their behavior (e.g. an explanation of changes it triggers in the parsing) should not be added to clang. Moreover, the pragmas should provide useful functionality to the user. For example, such functionality should address a practical use case and not be redundant i.e. cannot be achieved using existing features.

Note that some legacy extensions (published prior to OpenCL 3.0) still provide some non-conformant functionality for pragmas e.g. add diagnostics on the use of types or functions. This functionality is not guaranteed to remain in future releases. However, any future changes should not affect backward compatibility.

Address spaces attribute

Clang has arbitrary address space support using the `address_space(N)` attribute, where `N` is an integer number in the range specified in the Clang source code. This addresses spaces can be used along with the OpenCL address spaces however when such addresses spaces converted to/from OpenCL address spaces the behavior is not governed by OpenCL specification.

An OpenCL implementation provides a list of standard address spaces using keywords: `private`, `local`, `global`, and `generic`. In the AST and in the IR each of the address spaces will be represented by unique number provided in the Clang source code. The specific IDs for an address space do not have to match between the AST and the IR. Typically in the AST address space numbers represent logical segments while in the IR they represent physical segments. Therefore, machines with flat memory segments can map all AST address space numbers to the same physical segment ID or skip address space attribute completely while generating the IR. However, if the address space information is needed by the IR passes e.g. to improve alias analysis, it is recommended to keep it and only lower to reflect physical memory segments in the late machine passes. The mapping between logical and target address spaces is specified in the Clang's source code.

C++ for OpenCL Implementation Status

Clang implements language versions 1.0 and 2021 published in the official release of C++ for OpenCL Documentation.

Limited support of experimental C++ libraries is described in the experimental features.

GitHub issues for this functionality are typically prefixed with '[C++4OpenCL]' - click here to view the full bug list.

Missing features or with limited support

- Support of C++ for OpenCL 2021 is currently in experimental phase. Refer to OpenCL 3.0 status for details of common missing functionality from OpenCL 3.0.
- IR generation for non-trivial global destructors is incomplete (See: PR48047).
- Support of destrutors with non-default address spaces is incomplete (See: D109609).

OpenCL C 3.0 Usage

OpenCL C 3.0 language standard makes most OpenCL C 2.0 features optional. Optional functionality in OpenCL C 3.0 is indicated with the presence of feature-test macros (list of feature-test macros is here). Command-line flag -cl-ext can be used to override features supported by a target.

For cases when there is an associated extension for a specific feature (fp64 and 3d image writes) user should specify both (extension and feature) in command-line flag:

```
$ clang -cl-std=CL3.0 -cl-ext=+cl_khr_fp64,+__opencl_c_fp64 ...
$ clang -cl-std=CL3.0 -cl-ext=-cl_khr_fp64,-__opencl_c_fp64 ...
```

OpenCL C 3.0 Implementation Status

The following table provides an overview of features in OpenCL C 3.0 and their implementation status.

Category	Feature	Status	Reviews
Command line interface	New value for `-cl-std` flag	done	https://reviews.llvm.org/D88300
Predefined macros	New version macro	done	https://reviews.llvm.org/D88300
Predefined macros	Feature macros	done	https://reviews.llvm.org/D95776
Feature optionality	Generic address space	done	https://reviews.llvm.org/D95778 and https://reviews.llvm.org/D103401
Feature optionality	Builtin function overloads with generic address space	done	https://reviews.llvm.org/D105526, https://reviews.llvm.org/D107769
Feature optionality	Program scope variables in global memory	done	https://reviews.llvm.org/D103191
Feature optionality	3D image writes including builtin functions	done	https://reviews.llvm.org/D106260 (frontend)
Feature optionality	read_write images including builtin functions	done	https://reviews.llvm.org/D104915 (frontend) and https://reviews.llvm.org/D107539, https://reviews.llvm.org/D117899 (functions)
Feature optionality	C11 atomics memory scopes, ordering and builtin function	done	https://reviews.llvm.org/D106111, https://reviews.llvm.org/D119420

Category	Feature	Status	Reviews
Feature optionality	Blocks and Device-side kernel enqueue including builtin functions	done	https://reviews.llvm.org/D115640, https://reviews.llvm.org/D118605
Feature optionality	Pipes including builtin functions	done	https://reviews.llvm.org/D107154 (frontend) and https://reviews.llvm.org/D105858 (functions)
Feature optionality	Work group collective builtin functions	done	https://reviews.llvm.org/D105858
Feature optionality	Image types and builtin functions	done	https://reviews.llvm.org/D103911 (frontend) and https://reviews.llvm.org/D107539 (functions)
Feature optionality	Double precision floating point type	done	https://reviews.llvm.org/D96524
New functionality	RGBA vector components	done	https://reviews.llvm.org/D99969
New functionality	Subgroup functions	done	https://reviews.llvm.org/D105858, https://reviews.llvm.org/D118999
New functionality	Atomic mem scopes: subgroup, all devices including functions	done	https://reviews.llvm.org/D103241

Experimental features

Clang provides the following new WIP features for the developers to experiment and provide early feedback or contribute with further improvements. Feel free to contact us on cfe-dev or file a GitHub issue.

C++ libraries for OpenCL

There is ongoing work to support C++ standard libraries from LLVM's libcxx in OpenCL kernel code using C++ for OpenCL mode.

It is currently possible to include *type_traits* from C++17 in the kernel sources when the following clang extensions are enabled `__cl_clang_function_pointers` and `__cl_clang_variadic_functions`, see Clang Language Extensions for more details. The use of non-conformant features enabled by the extensions does not expose non-conformant behavior beyond the compilation i.e. does not get generated in IR or binary. The extension only appear in metaprogramming mechanism to identify or verify the properties of types. This allows to provide the full C++ functionality without a loss of portability. To avoid unsafe use of the extensions it is recommended that the extensions are disabled directly after the header include.

Example of Use:

The example of kernel code with *type_traits* is illustrated here.

```
#pragma OPENCL EXTENSION __cl_clang_function_pointers : enable
#pragma OPENCL EXTENSION __cl_clang_variadic_functions : enable
#include <type_traits>
#pragma OPENCL EXTENSION __cl_clang_function_pointers : disable
#pragma OPENCL EXTENSION __cl_clang_variadic_functions : disable

using sint_type = std::make_signed<unsigned int>::type;

__kernel void foo() {
  static_assert(!std::is_same<sint_type, unsigned int>::value);
}
```

The possible clang invocation to compile the example is as follows:

```
$ clang -I<path to libcxx checkout or installation>/include test.clcpp
```

Note that *type_traits* is a header only library and therefore no extra linking step against the standard libraries is required. See full example in Compiler Explorer.

More OpenCL specific C++ library implementations built on top of libcxx are available in libclcxx project.

OpenMP Support 729
General improvements 729
Cuda devices support 729
Directives execution modes 729
Data-sharing modes 730
Features not supported or with limited support for Cuda devices 730
OpenMP 5.0 Implementation Details 730
OpenMP 5.1 Implementation Details 733
OpenMP Extensions 735

OpenMP Support

Clang fully supports OpenMP 4.5. Clang supports offloading to X86_64, AArch64, PPC64[LE] and has basic support for Cuda devices.

- #pragma omp declare simd: Partial. We support parsing/semantic analysis + generation of special attributes for X86 target, but still missing the LLVM pass for vectorization.

In addition, the LLVM OpenMP runtime *libomp* supports the OpenMP Tools Interface (OMPT) on x86, x86_64, AArch64, and PPC64 on Linux, Windows, and macOS.

For the list of supported features from OpenMP 5.0 see OpenMP implementation details.

General improvements

- New collapse clause scheme to avoid expensive remainder operations. Compute loop index variables after collapsing a loop nest via the collapse clause by replacing the expensive remainder operation with multiplications and additions.
- The default schedules for the *distribute* and *for* constructs in a parallel region and in SPMD mode have changed to ensure coalesced accesses. For the *distribute* construct, a static schedule is used with a chunk size equal to the number of threads per team (default value of threads or as specified by the *thread_limit* clause if present). For the *for* construct, the schedule is static with chunk size of one.
- Simplified SPMD code generation for *distribute parallel for* when the new default schedules are applicable.
- When using the collapse clause on a loop nest the default behavior is to automatically extend the representation of the loop counter to 64 bits for the cases where the sizes of the collapsed loops are not known at compile time. To prevent this conservative choice and use at most 32 bits, compile your program with the *-fopenmp-optimistic-collapse*.

Cuda devices support

Directives execution modes

Clang code generation for target regions supports two modes: the SPMD and non-SPMD modes. Clang chooses one of these two modes automatically based on the way directives and clauses on those directives are used. The SPMD mode uses a simplified set of runtime functions thus increasing performance at the cost of supporting some OpenMP features. The non-SPMD mode is the most generic mode and supports all currently available OpenMP features. The compiler will always attempt to use the SPMD mode wherever possible. SPMD mode will not be used if:

- The target region contains user code (other than OpenMP-specific directives) in between the *target* and the *parallel* directives.

Data-sharing modes

Clang supports two data-sharing models for Cuda devices: *Generic* and *Cuda* modes. The default mode is *Generic*. *Cuda* mode can give an additional performance and can be activated using the *-fopenmp-cuda-mode* flag. In *Generic* mode all local variables that can be shared in the parallel regions are stored in the global memory. In *Cuda* mode local variables are not shared between the threads and it is user responsibility to share the required data between the threads in the parallel regions.

Features not supported or with limited support for Cuda devices

- Cancellation constructs are not supported.
- Doacross loop nest is not supported.
- User-defined reductions are supported only for trivial types.
- Nested parallelism: inner parallel regions are executed sequentially.
- Automatic translation of math functions in target regions to device-specific math functions is not implemented yet.
- Debug information for OpenMP target regions is supported, but sometimes it may be required to manually specify the address class of the inspected variables. In some cases the local variables are actually allocated in the global memory, but the debug info may be not aware of it.

OpenMP 5.0 Implementation Details

The following table provides a quick overview over various OpenMP 5.0 features and their implementation status. Please contact *openmp-dev* at *lists.llvm.org* for more information or if you want to help with the implementation.

Category	Feature	Status	Reviews
loop extension	support != in the canonical loop form	done	D54441
loop extension	#pragma omp loop (directive)	worked on	
loop extension	collapse imperfectly nested loop	done	
loop extension	collapse non-rectangular nested loop	done	
loop extension	C++ range-base for loop	done	
loop extension	clause: if for SIMD directives	done	
loop extension	inclusive scan extension (matching C++17 PSTL)	done	
memory management	memory allocators	done	r341687,r357929
memory management	allocate directive and allocate clause	done	r355614,r335952
OMPD	OMPD interfaces	not upstream	https://github.com/OpenMPToolsInterface/LLVM-openmp/tree/ompd-tests
OMPT	OMPT interfaces	mostly done	
thread affinity extension	thread affinity extension	done	
task extension	taskloop reduction	done	
task extension	task affinity	not upstream	https://github.com/jklinkenberg/openmp/tree/task-affinity
task extension	clause: depend on the taskwait construct	mostly done	D113540 (regular codegen only)

Category	Feature	Status	Reviews
task extension	depend objects and detachable tasks	done	
task extension	mutexinoutset dependence-type for tasks	done	D53380,D57576
task extension	combined taskloop constructs	done	
task extension	master taskloop	done	
task extension	parallel master taskloop	done	
task extension	master taskloop simd	done	
task extension	parallel master taskloop simd	done	
SIMD extension	atomic and simd constructs inside SIMD code	done	
SIMD extension	SIMD nontemporal	done	
device extension	infer target functions from initializers	worked on	
device extension	infer target variables from initializers	worked on	
device extension	OMP_TARGET_OFFLOAD environment variable	done	D50522
device extension	support full ‘defaultmap’ functionality	done	D69204
device extension	device specific functions	done	
device extension	clause: device_type	done	
device extension	clause: extended device	done	
device extension	clause: uses_allocators clause	done	
device extension	clause: in_reduction	worked on	r308768
device extension	omp_get_device_num()	worked on	D54342
device extension	structure mapping of references	unclaimed	
device extension	nested target declare	done	D51378
device extension	implicitly map ‘this’ (this[:1])	done	D55982
device extension	allow access to the reference count (omp_target_is_present)	done	
device extension	requires directive	partial	
device extension	clause: unified_shared_memory	done	D52625,D52359

Category	Feature	Status	Reviews
device extension	clause: unified_address	partial	
device extension	clause: reverse_offload	unclaimed parts	D52780
device extension	clause: atomic_default_mem_order	done	D53513
device extension	clause: dynamic_allocators	unclaimed parts	D53079
device extension	user-defined mappers	worked on	D56326,D58638,D58523,D58074,D60972,D59474
device extension	mapping lambda expression	done	D51107
device extension	clause: use_device_addr for target data	done	
device extension	support close modifier on map clause	done	D55719,D55892
device extension	teams construct on the host device	done	r371553
device extension	support non-contiguous array sections for target update	done	
device extension	pointer attachment	unclaimed	
device extension	map clause reordering based on map types	unclaimed	
atomic extension	hints for the atomic construct	done	D51233
base language	C11 support	done	
base language	C++11/14/17 support	done	
base language	lambda support	done	
misc extension	array shaping	done	D74144
misc extension	library shutdown (omp_pause_resource[_all])	unclaimed parts	D55078
misc extension	metadirectives	worked on	D91944
misc extension	conditional modifier for lastprivate clause	done	
misc extension	iterator and multidependences	done	
misc extension	depobj directive and depobj dependency kind	done	
misc extension	user-defined function variants	worked on	D67294, D64095, D71847, D71830, D109635
misc extension	pointer/reference to pointer based array reductions	unclaimed	
misc extension	prevent new type definitions in clauses	done	
memory model extension	memory model update (seq_cst, acq_rel, release, acquire,…)	done	

OpenMP 5.1 Implementation Details

The following table provides a quick overview over various OpenMP 5.1 features and their implementation status, as defined in the technical report 8 (TR8). Please contact *openmp-dev* at *lists.llvm.org* for more information or if you want to help with the implementation.

Category	Feature	Status	Reviews
atomic extension	'compare' clause on atomic construct	worked on	
atomic extension	'fail' clause on atomic construct	worked on	
base language	C++ attribute specifier syntax	done	D105648
device extension	'present' map type modifier	done	D83061, D83062, D84422
device extension	'present' motion modifier	done	D84711, D84712
device extension	'present' in defaultmap clause	done	D92427
device extension	map clause reordering reordering based on 'present' modifier	unclaimed	
device extension	device-specific environment variables	unclaimed	
device extension	omp_target_is_accessible routine	unclaimed	
device extension	omp_get_mapped_ptr routine	unclaimed	
device extension	new async target memory copy routines	unclaimed	
device extension	thread_limit clause on target construct	unclaimed	
device extension	has_device_addr clause on target construct	unclaimed	
device extension	iterators in map clause or motion clauses	unclaimed	
device extension	indirect clause on declare target directive	unclaimed	
device extension	allow virtual functions calls for mapped object on device	unclaimed	
device extension	interop construct	partial	parsing/sema done: D98558, D98834, D98815
device extension	assorted routines for querying interoperable properties	unclaimed	
loop extension	Loop tiling transformation	done	D76342
loop extension	Loop unrolling transformation	done	D99459
loop extension	'reproducible'/'unconstrained' modifiers in 'order' clause	unclaimed	
memory management	alignment extensions for allocate directive and clause	worked on	

Category	Feature	Status	Reviews
memory management	new memory management routines	unclaimed	
memory management	changes to omp_alloctrait_key enum	unclaimed	
memory model extension	seq_cst clause on flush construct	unclaimed	
misc extension	‘omp_all_memory’ keyword and use in ‘depend’ clause	unclaimed	
misc extension	error directive	unclaimed	
misc extension	scope construct	unclaimed	
misc extension	routines for controlling and querying team regions	unclaimed	
misc extension	changes to ompt_scope_endpoint_t enum	unclaimed	
misc extension	omp_display_env routine	unclaimed	
misc extension	extended OMP_PLACES syntax	unclaimed	
misc extension	OMP_NUM_TEAMS and OMP_TEAMS_THREAD_LIMIT env vars	unclaimed	
misc extension	‘target_device’ selector in context specifier	unclaimed	
misc extension	begin/end declare variant	done	D71179
misc extension	dispatch construct and function variant argument adjustment	worked on	D99537, D99679
misc extension	assume and assumes directives	worked on	
misc extension	nothing directive	worked on	
misc extension	masked construct and related combined constructs	worked on	D99995, D100514
misc extension	default(firstprivate) & default(private)	partial	firstprivate done: D75591
other	deprecating master construct	unclaimed	
OMPT	new barrier types added to ompt_sync_region_t enum	unclaimed	
OMPT	async data transfers added to ompt_target_data_op_t enum	unclaimed	
OMPT	new barrier state values added to ompt_state_t enum	unclaimed	
OMPT	new ‘emi’ callbacks for external monitoring interfaces	unclaimed	
task extension	‘strict’ modifier for taskloop construct	unclaimed	
task extension	inoutset in depend clause	unclaimed	
task extension	nowait clause on taskwait	unclaimed	

OpenMP Extensions

The following table provides a quick overview over various OpenMP extensions and their implementation status. These extensions are not currently defined by any standard, so links to associated LLVM documentation are provided. As these extensions mature, they will be considered for standardization. Please contact *openmp-dev* at *lists.llvm.org* to provide feedback.

Category	Feature	Status	Reviews
atomic extension	'atomic' strictly nested within 'teams'	prototyped	D126323
device extension	'ompx_hold' map type modifier	prototyped	D106509, D106510

SYCL Compiler and Runtime architecture design

Introduction 735

Address space handling 735

Introduction

This document describes the architecture of the SYCL compiler and runtime library. More details are provided in external document, which are going to be added to clang documentation in the future.

Address space handling

The SYCL specification represents pointers to disjoint memory regions using C++ wrapper classes on an accelerator to enable compilation with a standard C++ toolchain and a SYCL compiler toolchain. Section 3.8.2 of SYCL 2020 specification defines memory model, section 4.7.7 - address space classes and section 5.9 covers address space deduction. The SYCL specification allows two modes of address space deduction: "generic as default address space" (see section 5.9.3) and "inferred address space" (see section 5.9.4). Current implementation supports only "generic as default address space" mode.

SYCL borrows its memory model from OpenCL however SYCL doesn't perform the address space qualifier inference as detailed in OpenCL C v3.0 6.7.8.

The default address space is "generic-memory", which is a virtual address space that overlaps the global, local, and private address spaces. SYCL mode enables following conversions:

- explicit conversions to/from the default address space from/to the address space-attributed type
- implicit conversions from the address space-attributed type to the default address space
- explicit conversions to/from the global address space from/to the `__attribute__((opencl_global_device))` or `__attribute__((opencl_global_host))` address space-attributed type
- implicit conversions from the `__attribute__((opencl_global_device))` or `__attribute__((opencl_global_host))` address space-attributed type to the global address space

All named address spaces are disjoint and sub-sets of default address space.

The SPIR target allocates SYCL namespace scope variables in the global address space.

Pointers to default address space should get lowered into a pointer to a generic address space (or flat to reuse more general terminology). But depending on the allocation context, the default address space of a non-pointer type is assigned to a specific address space. This is described in common address space deduction rules section.

This is also in line with the behaviour of CUDA (small example).

`multi_ptr` class implementation example:

```
// check that SYCL mode is ON and we can use non-standard decorations
#if defined(__SYCL_DEVICE_ONLY__)
// GPU/accelerator implementation
```

```
template <typename T, address_space AS> class multi_ptr {
  // DecoratedType applies corresponding address space attribute to the type T
  // DecoratedType<T, global_space>::type == "__attribute__((opencl_global)) T"
  // See sycl/include/CL/sycl/access/access.hpp for more details
  using pointer_t = typename DecoratedType<T, AS>::type *;

  pointer_t m_Pointer;
  public:
  pointer_t get() { return m_Pointer; }
  T& operator* () { return *reinterpret_cast<T*>(m_Pointer); }
}
#else
// CPU/host implementation
template <typename T, address_space AS> class multi_ptr {
  T *m_Pointer; // regular undecorated pointer
  public:
  T *get() { return m_Pointer; }
  T& operator* () { return *m_Pointer; }
}
#endif
```

Depending on the compiler mode, `multi_ptr` will either decorate its internal data with the address space attribute or not.

To utilize clang's existing functionality, we reuse the following OpenCL address space attributes for pointers:

Address space attribute	SYCL address_space enumeration
`__attribute__((opencl_global))`	global_space, constant_space
`__attribute__((opencl_global_device))`	global_space
`__attribute__((opencl_global_host))`	global_space
`__attribute__((opencl_local))`	local_space
`__attribute__((opencl_private))`	private_space

```
//TODO: add support for __attribute__((opencl_global_host)) and __attribute__((opencl_global_device)).
```

HLSL Support

Introduction 737
Project Goals 737
Non-Goals 737
Guiding Principles 737
Architectural Direction 737
DXC Driver 738
Parser 738
Sema 738
CodeGen 738
HLSL Language 738
An Aside on GPU Languages 738
Pointers & References 739
HLSL `this` Keyword 739
Bitshifts 739
Non-short Circuiting Logical Operators 739
Precise Qualifier 739
Differences in Templates 739
Vector Extensions 739
Standard Library 739
Unsupported C & C++ Features 739

Introduction

HLSL Support is under active development in the Clang codebase. This document describes the high level goals of the project, the guiding principles, as well as some idiosyncrasies of the HLSL language and how we intend to support them in Clang.

Project Goals

The long term goal of this project is to enable Clang to function as a replacement for the DirectXShaderCompiler (DXC) in all its supported use cases. Accomplishing that goal will require Clang to be able to process most existing HLSL programs with a high degree of source compatibility.

Non-Goals

HLSL ASTs do not need to be compatible between DXC and Clang. We do not expect identical code generation or that features will resemble DXC's implementation or architecture. In fact, we explicitly expect to deviate from DXC's implementation in key ways.

Guiding Principles

This document lacks details for architectural decisions that are not yet finalized. Our top priorities are quality, maintainability, and flexibility. In accordance with community standards we are expecting a high level of test coverage, and we will engineer our solutions with long term maintenance in mind. We are also working to limit modifications to the Clang C++ code paths and share as much functionality as possible.

Architectural Direction

HLSL support in Clang is expressed as C++ minus unsupported C and C++ features. This is different from how other Clang languages are implemented. Most languages in Clang are additive on top of C.

HLSL is not a formally or fully specified language, and while our goals require a high level of source compatibility, implementations can vary and we have some flexibility to be more or less permissive in some cases. For modern HLSL DXC is the reference implementation.

The HLSL effort prioritizes following similar patterns for other languages, drivers, runtimes and targets. Specifically, We will maintain separation between HSLS-specific code and the rest of Clang as much as possible following patterns in use in Clang code today (i.e. ParseHLSL.cpp, SemaHLSL.cpp, CGHLSL*.cpp…). We will use inline checks on language options where the code is simple and isolated, and prefer HLSL-specific implementation files for any code of reasonable complexity.

In places where the HLSL language is in conflict with C and C++, we will seek to make minimally invasive changes guarded under the HLSL language options. We will seek to make HLSL language support as minimal a maintenance burden as possible.

DXC Driver

A DXC driver mode will provide command-line compatibility with DXC, supporting DXC's options and flags. The DXC driver is HLSL-specific and will create an HLSLToolchain which will provide the basis to support targeting both DirectX and Vulkan.

Parser

Following the examples of other parser extensions HLSL will add a ParseHLSL.cpp file to contain the implementations of HLSL-specific extensions to the Clang parser. The HLSL grammar shares most of its structure with C and C++, so we will use the existing C/C++ parsing code paths.

Sema

HLSL's Sema implementation will also provide an `ExternalSemaSource`. In DXC, an `ExternalSemaSource` is used to provide definitions for HLSL built-in data types and built-in templates. Clang is already designed to allow an attached `ExternalSemaSource` to lazily complete data types, which is a **huge** performance win for HLSL.

CodeGen

Like OpenCL, HLSL relies on capturing a lot of information into IR metadata. *hand wave hand wave hand wave* As a design principle here we want our IR to be idiomatic Clang IR as much as possible. We will use IR attributes wherever we can, and use metadata as sparingly as possible. One example of a difference from DXC already implemented in Clang is the use of target triples to communicate shader model versions and shader stages.

Our HLSL CodeGen implementation should also have an eye toward generating IR that will map directly to targets other than DXIL. While IR itself is generally not re-targetable, we want to share the Clang CodeGen implementation for HLSL with other GPU graphics targets like SPIR-V and possibly other GPU and even CPU targets.

HLSL Language

The HLSL language is insufficiently documented, and not formally specified. Documentation is available on Microsoft's website. The language syntax is similar enough to C and C++ that carefully written C and C++ code is valid HLSL. HLSL has some key differences from C & C++ which we will need to handle in Clang.

HLSL is not a conforming or valid extension or superset of C or C++. The language has key incompatibilities with C and C++, both syntactically and semantically.

An Aside on GPU Languages

Due to HLSL being a GPU targeted language HLSL is a Single Program Multiple Data (SPMD) language relying on the implicit parallelism provided by GPU hardware. Some language features in HLSL enable programmers to take advantage of the parallel nature of GPUs in a hardware abstracted language.

HLSL also prohibits some features of C and C++ which can have catastrophic performance or are not widely supportable on GPU hardware or drivers. As an example, register spilling is often excessively expensive on GPUs, so HLSL requires all functions to be inlined during code generation, and does not support a runtime calling convention.

Pointers & References

HLSL does not support referring to values by address. Semantically all variables are value-types and behave as such. HLSL disallows the pointer dereference operators (unary `*`, and `->`), as well as the address of operator (unary `&`). While HLSL disallows pointers and references in the syntax, HLSL does use reference types in the AST, and we intend to use pointer decay in the AST in the Clang implementation.

HLSL `this` Keyword

HLSL does support member functions, and (in HLSL 2021) limited operator overloading. With member function support, HLSL also has a `this` keyword. The `this` keyword is an example of one of the places where HLSL relies on references in the AST, because `this` is a reference.

Bitshifts

In deviation from C, HLSL bitshifts are defined to mask the shift count by the size of the type. In DXC, the semantics of LLVM IR were altered to accommodate this, in Clang we intend to generate the mask explicitly in the IR. In cases where the shift value is constant, this will be constant folded appropriately, in other cases we can clean it up in the DXIL target.

Non-short Circuiting Logical Operators

In HLSL 2018 and earlier, HLSL supported logical operators (and the ternary operator) on vector types. This behavior required that operators not short circuit. The non-short circuiting behavior applies to all data types until HLSL 2021. In HLSL 2021, logical and ternary operators do not support vector types instead builtin functions `and`, `or` and `select` are available, and operators short circuit matching C behavior.

Precise Qualifier

HLSL has a `precise` qualifier that behaves unlike anything else in the C language. The support for this qualifier in DXC is buggy, so our bar for compatibility is low.

The `precise` qualifier applies in the inverse direction from normal qualifiers. Rather than signifying that the declaration containing `precise` qualifier be precise, it signifies that the operations contributing to the declaration's value be `precise`. Additionally, `precise` is a misnomer: values attributed as `precise` comply with IEEE-754 floating point semantics, and are prevented from optimizations which could decrease *or increase* precision.

Differences in Templates

HLSL uses templates to define builtin types and methods, but disallowed user-defined templates until HLSL 2021. HLSL also allows omitting empty template parameter lists when all template parameters are defaulted. This is an ambiguous syntax in C++, but Clang detects the case and issues a diagnostic. This makes supporting the case in Clang minimally invasive.

Vector Extensions

HLSL uses the OpenCL vector extensions, and also provides C++-style constructors for vectors that are not supported by Clang.

Standard Library

HLSL does not support the C or C++ standard libraries. Like OpenCL, HLSL describes its own library of built in types, complex data types, and functions.

Unsupported C & C++ Features

HLSL does not support all features of C and C++. In implementing HLSL in Clang use of some C and C++ features will produce diagnostics under HLSL, and others will be supported as language extensions. In general, any C or C++ feature that can be supported by the DXIL and SPIR-V code generation targets could be treated as a clang HLSL extension. Features that cannot be lowered to DXIL or SPIR-V, must be diagnosed as errors.

HLSL does not support the following C features:

- Pointers
- References
- `goto` or labels
- Variable Length Arrays
- `_Complex` and `_Imaginary`
- C Threads or Atomics (or Obj-C blocks)
- `union` types (in progress for HLSL 202x)
- Most features C11 and later

HLSL does not support the following C++ features:

- RTTI
- Exceptions
- Multiple inheritance
- Access specifiers
- Anonymous or inline namespaces
- `new` & `delete` operators in all of their forms (array, placement, etc)
- Constructors and destructors
- Any use of the `virtual` keyword
- Most features C++11 and later

ThinLTO

Introduction **740**

Current Status **741**

Clang/LLVM 741

Linkers 741

Usage **741**

Basic 741

Controlling Backend Parallelism 741

Incremental 742

Cache Pruning 742

Clang Bootstrap 742

More Information **743**

Introduction

ThinLTO compilation is a new type of LTO that is both scalable and incremental. *LTO* (Link Time Optimization) achieves better runtime performance through whole-program analysis and cross-module optimization. However, monolithic LTO implements this by merging all input into a single module, which is not scalable in time or memory, and also prevents fast incremental compiles.

In ThinLTO mode, as with regular LTO, clang emits LLVM bitcode after the compile phase. The ThinLTO bitcode is augmented with a compact summary of the module. During the link step, only the summaries are read and merged into a combined summary index, which includes an index of function locations for later cross-module function importing. Fast and efficient whole-program analysis is then performed on the combined summary index.

However, all transformations, including function importing, occur later when the modules are optimized in fully parallel backends. By default, linkers that support ThinLTO are set up to launch the ThinLTO backends in threads.

So the usage model is not affected as the distinction between the fast serial thin link step and the backends is transparent to the user.

For more information on the ThinLTO design and current performance, see the LLVM blog post ThinLTO: Scalable and Incremental LTO. While tuning is still in progress, results in the blog post show that ThinLTO already performs well compared to LTO, in many cases matching the performance improvement.

Current Status

Clang/LLVM

The 3.9 release of clang includes ThinLTO support. However, ThinLTO is under active development, and new features, improvements and bugfixes are being added for the next release. For the latest ThinLTO support, build a recent version of clang and LLVM.

Linkers

ThinLTO is currently supported for the following linkers:

- **gold (via the gold-plugin)**: Similar to monolithic LTO, this requires using a gold linker configured with plugins enabled.
- **ld64**: Starting with Xcode 8.
- **lld**: Starting with r284050 for ELF, r298942 for COFF.

Usage

Basic

To utilize ThinLTO, simply add the -flto=thin option to compile and link. E.g.

```
% clang -flto=thin -O2 file1.c file2.c -c
% clang -flto=thin -O2 file1.o file2.o -o a.out
```

When using lld-link, the -flto option need only be added to the compile step:

```
% clang-cl -flto=thin -O2 -c file1.c file2.c
% lld-link /out:a.exe file1.obj file2.obj
```

As mentioned earlier, by default the linkers will launch the ThinLTO backend threads in parallel, passing the resulting native object files back to the linker for the final native link. As such, the usage model is the same as non-LTO.

With gold, if you see an error during the link of the form:

```
/usr/bin/ld: error: /path/to/clang/bin/../lib/LLVMgold.so: could not load plugin library: /path/to/clang/bin/../lib/LLVMgold.so: cannot open shared object file: No such file or directory
```

Then either gold was not configured with plugins enabled, or clang was not built with `-DLLVM_BINUTILS_INCDIR` set properly. See the instructions for the LLVM gold plugin.

Controlling Backend Parallelism

By default, the ThinLTO link step will launch as many threads in parallel as there are cores. If the number of cores can't be computed for the architecture, then it will launch `std::thread::hardware_concurrency` number of threads in parallel. For machines with hyper-threading, this is the total number of virtual cores. For some applications and machine configurations this may be too aggressive, in which case the amount of parallelism can be reduced to `N` via:

- gold: `-Wl,-plugin-opt,jobs=N`
- ld64: `-Wl,-mllvm,-threads=N`
- lld: `-Wl,--thinlto-jobs=N`
- lld-link: `/opt:lldltojobs=N`

Other possible values for `N` are:

- 0: Use one thread per physical core (default)
- 1: Use a single thread only (disable multi-threading)
- all: Use one thread per logical core (uses all hyper-threads)

Incremental

ThinLTO supports fast incremental builds through the use of a cache, which currently must be enabled through a linker option.

- gold (as of LLVM 4.0): `-Wl,-plugin-opt,cache-dir=/path/to/cache`
- ld64 (support in clang 3.9 and Xcode 8): `-Wl,-cache_path_lto,/path/to/cache`
- ELF lld (as of LLVM 5.0): `-Wl,--thinlto-cache-dir=/path/to/cache`
- COFF lld-link (as of LLVM 6.0): `/lldltocache:/path/to/cache`

Cache Pruning

To help keep the size of the cache under control, ThinLTO supports cache pruning. Cache pruning is supported with gold, ld64 and ELF and COFF lld, but currently only gold, ELF and COFF lld allow you to control the policy with a policy string. The cache policy must be specified with a linker option.

- gold (as of LLVM 6.0): `-Wl,-plugin-opt,cache-policy=POLICY`
- ELF lld (as of LLVM 5.0): `-Wl,--thinlto-cache-policy,POLICY`
- COFF lld-link (as of LLVM 6.0): `/lldltocachepolicy:POLICY`

A policy string is a series of key-value pairs separated by `:` characters. Possible key-value pairs are:

- `cache_size=X%`: The maximum size for the cache directory is `X` percent of the available space on the disk. Set to 100 to indicate no limit, 50 to indicate that the cache size will not be left over half the available disk space. A value over 100 is invalid. A value of 0 disables the percentage size-based pruning. The default is 75%.
- `cache_size_bytes=X`, `cache_size_bytes=Xk`, `cache_size_bytes=Xm`, `cache_size_bytes=Xg`: Sets the maximum size for the cache directory to `X` bytes (or KB, MB, GB respectively). A value over the amount of available space on the disk will be reduced to the amount of available space. A value of 0 disables the byte size-based pruning. The default is no byte size-based pruning.

  Note that ThinLTO will apply both size-based pruning policies simultaneously, and changing one does not affect the other. For example, a policy of `cache_size_bytes=1g` on its own will cause both the 1GB and default 75% policies to be applied unless the default `cache_size` is overridden.
- `cache_size_files=X`: Set the maximum number of files in the cache directory. Set to 0 to indicate no limit. The default is 1000000 files.
- `prune_after=Xs`, `prune_after=Xm`, `prune_after=Xh`: Sets the expiration time for cache files to `X` seconds (or minutes, hours respectively). When a file hasn't been accessed for `prune_after` seconds, it is removed from the cache. A value of 0 disables the expiration-based pruning. The default is 1 week.
- `prune_interval=Xs`, `prune_interval=Xm`, `prune_interval=Xh`: Sets the pruning interval to `X` seconds (or minutes, hours respectively). This is intended to be used to avoid scanning the directory too often. It does not impact the decision of which files to prune. A value of 0 forces the scan to occur. The default is every 20 minutes.

Clang Bootstrap

To bootstrap clang/LLVM with ThinLTO, follow these steps:

1. The host compiler must be a version of clang that supports ThinLTO.
2. The host linker must support ThinLTO (and in the case of gold, must be configured with plugins enabled).
3. Use the following additional CMake variables when configuring the bootstrap compiler build:
   - `-DLLVM_ENABLE_LTO=Thin`

- `-DCMAKE_C_COMPILER=/path/to/host/clang`
- `-DCMAKE_CXX_COMPILER=/path/to/host/clang++`
- `-DCMAKE_RANLIB=/path/to/host/llvm-ranlib`
- `-DCMAKE_AR=/path/to/host/llvm-ar`

Or, on Windows:

- `-DLLVM_ENABLE_LTO=Thin`
- `-DCMAKE_C_COMPILER=/path/to/host/clang-cl.exe`
- `-DCMAKE_CXX_COMPILER=/path/to/host/clang-cl.exe`
- `-DCMAKE_LINKER=/path/to/host/lld-link.exe`
- `-DCMAKE_RANLIB=/path/to/host/llvm-ranlib.exe`
- `-DCMAKE_AR=/path/to/host/llvm-ar.exe`

1. To use additional linker arguments for controlling the backend parallelism or enabling incremental builds of the bootstrap compiler, after configuring the build, modify the resulting CMakeCache.txt file in the build directory. Specify any additional linker options after `CMAKE_EXE_LINKER_FLAGS:STRING=`. Note the configure may fail if linker plugin options are instead specified directly in the previous step.

The `BOOTSTRAP_LLVM_ENABLE_LTO=Thin` will enable ThinLTO for stage 2 and stage 3 in case the compiler used for stage 1 does not support the ThinLTO option.

More Information

- From LLVM project blog: ThinLTO: Scalable and Incremental LTO

API Notes: Annotations Without Modifying Headers

The Problem: You have headers you want to use, but you also want to add extra information to the API. You don't want to put that information in the headers themselves — perhaps because you want to keep them clean for other clients, or perhaps because they're from some open source project and you don't want to modify them at all.

Incomplete solution: Redeclare all the interesting parts of the API in your own header and add the attributes you want. Unfortunately, this:

- doesn't work with attributes that must be present on a definition
- doesn't allow changing the definition in other ways
- requires your header to be included in any client code to take effect

Better solution: Provide a "sidecar" file with the information you want to add, and have that automatically get picked up by the module-building logic in the compiler.

That's API notes.

API notes use a YAML-based file format. YAML is a format best explained by example, so here is a small example from the compiler test suite of API notes for a hypothetical "SomeKit" framework.

Usage

API notes files are found relative to the module map that defines a module, under the name "SomeKit.apinotes" for a module named "SomeKit". Additionally, a file named "SomeKit_private.apinotes" will also be picked up to go with a private module map. For bare modules these two files will be in the same directory as the corresponding module map; for framework modules, they should be placed in the Headers and PrivateHeaders directories, respectively. The module map for a private top-level framework module should be placed in the PrivateHeaders directory as well, though it does not need an additional "_private" suffix on its name.

Clang will search for API notes files next to module maps only when passed the `-fapi-notes-modules` option.

Limitations

- Since they're identified by module name, API notes cannot be used to modify arbitrary textual headers.

"Versioned" API Notes

Many API notes affect how a C API is imported into Swift. In order to change that behavior while still remaining backwards-compatible, API notes can be selectively applied based on the Swift compatibility version provided to the compiler (e.g. `-fapi-notes-swift-version=5`). The rule is that an explicitly-versioned API note applies to that version *and all earlier versions,* and any applicable explicitly-versioned API note takes precedence over an unversioned API note.

Reference

An API notes file contains a YAML dictionary with the following top-level entries:

Name: The name of the module (the framework name, for frameworks). Note that this is always the name of a top-level module, even within a private API notes file.

```
Name: MyFramework
```

Classes, Protocols, Tags, Typedefs, Globals, Enumerators, Functions: Arrays of top-level declarations. Each entry in the array must have a 'Name' key with its Objective-C name. "Tags" refers to structs, enums, and unions; "Enumerators" refers to enum cases.

```
Classes:
- Name: MyController
  …
- Name: MyView
  …
```

SwiftVersions: Contains explicit information for backwards compatibility. Each entry in the array contains a 'Version' key, which should be set to '4' for annotations that only apply to Swift 4 mode and earlier. The other entries in this dictionary are the same declaration entries as at the top level: Classes, Protocols, Tags, Typedefs, Globals, Enumerators, and Functions.

```
SwiftVersions:
- Version: 4
  Classes: …
  Protocols: …
```

Each entry under 'Classes' and 'Protocols' can contain "Methods" and "Properties" arrays, in addition to the attributes described below:

Methods: Identified by 'Selector' and 'MethodKind'; the MethodKind is either "Instance" or "Class".

```
Classes:
- Name: UIViewController
  Methods:
  - Selector: "presentViewController:animated:"
    MethodKind: Instance
    …
```

Properties: Identified by 'Name' and 'PropertyKind'; the PropertyKind is also either "Instance" or "Class".

```
Classes:
- Name: UIView
  Properties:
  - Name: subviews
    PropertyKind: Instance
    …
```

Each declaration supports the following annotations (if relevant to that declaration kind), all of which are optional:

SwiftName: Equivalent to `NS_SWIFT_NAME`. For a method, must include the full Swift name with all arguments. Use "_" to omit an argument label.

```
- Selector: "presentViewController:animated:"
  MethodKind: Instance
  SwiftName: "present(_:animated:)"

- Class: NSBundle
  SwiftName: Bundle
```

Availability, AvailabilityMsg: A value of "nonswift" is equivalent to `NS_SWIFT_UNAVAILABLE`. A value of "available" can be used in the "SwiftVersions" section to undo the effect of "nonswift".

```
- Selector: "dealloc"
  MethodKind: Instance
  Availability: nonswift
  AvailabilityMsg: "prefer 'deinit'"
```

SwiftPrivate: Equivalent to NS_REFINED_FOR_SWIFT.

```
- Name: CGColorEqualToColor
  SwiftPrivate: true
```

Nullability: Used for properties and globals. There are four options, identified by their initials:

- `Nonnull` or `N` (corresponding to `_Nonnull`)
- `Optional` or `O` (corresponding to `_Nullable`)
- `Unspecified` or `U` (corresponding to `_Null_unspecified`)
- `Scalar` or `S` (deprecated)

Note that 'Nullability' is overridden by 'Type', even in a "SwiftVersions" section.

> **Note**
>
> 'Nullability' can also be used to describe the argument types of methods and functions, but this usage is deprecated in favor of 'Parameters' (see below).

```
- Name: dataSource
  Nullability: O
```

NullabilityOfRet: Used for methods and functions. Describes the nullability of the return type.

Note that 'NullabilityOfRet' is overridden by 'ResultType', even in a "SwiftVersions" section.

> **Warning**
>
> Due to a compiler bug, 'NullabilityOfRet' may change nullability of the parameters as well (rdar://30544062). Avoid using it and instead use 'ResultType' and specify the return type along with a nullability annotation (see documentation for 'ResultType').

```
- Selector: superclass
  MethodKind: Class
  NullabilityOfRet: O
```

Type: Used for properties and globals. This completely overrides the type of the declaration; it should ideally only be used for Swift backwards compatibility, when existing type information has been made more precise in a header. Prefer ‘Nullability’ and other annotations when possible.

We parse the specified type as if it appeared at the location of the declaration whose type is being modified. Macros are not available and nullability must be applied explicitly (even in an `NS_ASSUME_NONNULL_BEGIN` section).

```
- Name: delegate
  PropertyKind: Instance
  Type: "id"
```

ResultType: Used for methods and functions. This completely overrides the return type; it should ideally only be used for Swift backwards compatibility, when existing type information has been made more precise in a header.

We parse the specified type as if it appeared at the location of the declaration whose type is being modified. Macros are not available and nullability must be applied explicitly (even in an `NS_ASSUME_NONNULL_BEGIN` section).

```
- Selector: "subviews"
  MethodKind: Instance
  ResultType: "NSArray * _Nonnull"
```

SwiftImportAsAccessors: Used for properties. If true, the property will be exposed in Swift as its accessor methods, rather than as a computed property using `var`.

```
- Name: currentContext
  PropertyKind: Class
  SwiftImportAsAccessors: true
```

NSErrorDomain: Used for `NSError` code enums. The value is the name of the associated domain `NSString` constant; an empty string (`""`) means the enum is a normal enum rather than an error code.

```
- Name: MKErrorCode
  NSErrorDomain: MKErrorDomain
```

SwiftWrapper: Controls `NS_STRING_ENUM` and `NS_EXTENSIBLE_STRING_ENUM`. There are three options:

- “struct” (extensible)
- “enum”
- “none”

Note that even an “enum” wrapper is still presented as a struct in Swift; it’s just a “more enum-like” struct.

```
- Name: AVMediaType
  SwiftWrapper: none
```

EnumKind: Has the same effect as `NS_ENUM` and `NS_OPTIONS`. There are four options:

- “NSEnum” / “CFEnum”
- “NSClosedEnum” / “CFClosedEnum”
- “NSOptions” / “CFOptions”
- “none”

```
- Name: GKPhotoSize
  EnumKind: none
```

Parameters: Used for methods and functions. Parameters are identified by a 0-based ‘Position’ and support the ‘Nullability’, ‘NoEscape’, and ‘Type’ keys.

> **Note**
>
> Using ‘Parameters’ within a parameter entry to describe the parameters of a block is not implemented. Use ‘Type’ on the entire parameter instead.

```
- Selector: "isEqual:"
  MethodKind: Instance
  Parameters:
  - Position: 0
    Nullability: O
```

NoEscape: Used only for block parameters. Equivalent to `NS_NOESCAPE`.

```
- Name: dispatch_sync
  Parameters:
  - Position: 0
    NoEscape: true
```

SwiftBridge: Used for Objective-C class types bridged to Swift value types. An empty string (“”) means a type is not bridged. Not supported outside of Apple frameworks (the Swift side of it requires conforming to implementation-detail protocols that are subject to change).

```
- Name: NSIndexSet
  SwiftBridge: IndexSet
```

DesignatedInit: Used for init methods. Equivalent to `NS_DESIGNATED_INITIALIZER`.

```
- Selector: "initWithFrame:"
  MethodKind: Instance
  DesignatedInit: true
```

Clang “man” pages

The following documents are command descriptions for all of the Clang tools. These pages describe how to use the Clang commands and what their options are. Note that these pages do not describe all of the options available for all tools. To get a complete listing, pass the `--help` (general options) or `--help-hidden` (general and debugging options) arguments to the tool you are interested in.

Basic Commands

clang - the Clang C, C++, and Objective-C compiler

SYNOPSIS

clang [*options*] *filename …*

DESCRIPTION

clang is a C, C++, and Objective-C compiler which encompasses preprocessing, parsing, optimization, code generation, assembly, and linking. Depending on which high-level mode setting is passed, Clang will stop before doing a full link. While Clang is highly integrated, it is important to understand the stages of compilation, to understand how to invoke it. These stages are:

Driver

The clang executable is actually a small driver which controls the overall execution of other tools such as the compiler, assembler and linker. Typically you do not need to interact with the driver, but you transparently use it to run the other tools.

Preprocessing

This stage handles tokenization of the input source file, macro expansion, #include expansion and handling of other preprocessor directives. The output of this stage is typically called a “.i” (for C), “.ii” (for C++), “.mi” (for Objective-C), or “.mii” (for Objective-C++) file.

Parsing and Semantic Analysis

This stage parses the input file, translating preprocessor tokens into a parse tree. Once in the form of a parse tree, it applies semantic analysis to compute types for expressions as well and determine whether the code is well formed. This stage is responsible for generating most of the compiler warnings as well as parse errors. The output of this stage is an “Abstract Syntax Tree” (AST).

Code Generation and Optimization

This stage translates an AST into low-level intermediate code (known as “LLVM IR”) and ultimately to machine code. This phase is responsible for optimizing the generated code and handling target-specific code generation. The output of this stage is typically called a “.s” file or “assembly” file.

Clang also supports the use of an integrated assembler, in which the code generator produces object files directly. This avoids the overhead of generating the “.s” file and of calling the target assembler.

Assembler

This stage runs the target assembler to translate the output of the compiler into a target object file. The output of this stage is typically called a “.o” file or “object” file.

Linker

This stage runs the target linker to merge multiple object files into an executable or dynamic library. The output of this stage is typically called an “a.out”, “.dylib” or “.so” file.

Clang Static Analyzer

The Clang Static Analyzer is a tool that scans source code to try to find bugs through code analysis. This tool uses many parts of Clang and is built into the same driver. Please see <https://clang-analyzer.llvm.org> for more details on how to use the static analyzer.

OPTIONS

Stage Selection Options

-E
Run the preprocessor stage.

-fsyntax-only
Run the preprocessor, parser and type checking stages.

-S
Run the previous stages as well as LLVM generation and optimization stages and target-specific code generation, producing an assembly file.

-c
Run all of the above, plus the assembler, generating a target “.o” object file.

no `stage selection option`
If no stage selection option is specified, all stages above are run, and the linker is run to combine the results into an executable or shared library.

Language Selection and Mode Options

-x `<language>`
Treat subsequent input files as having type language.

-std=`<standard>`
Specify the language standard to compile for.
Supported values for the C language are:

```
c89
c90
```

`iso9899:1990`

ISO C 1990

`iso9899:199409`

ISO C 1990 with amendment 1

`gnu89`
`gnu90`

ISO C 1990 with GNU extensions

`c99`
`iso9899:1999`

ISO C 1999

`gnu99`

ISO C 1999 with GNU extensions

`c11`
`iso9899:2011`

ISO C 2011

`gnu11`

ISO C 2011 with GNU extensions

`c17`
`iso9899:2017`

ISO C 2017

`gnu17`

ISO C 2017 with GNU extensions

The default C language standard is `gnu17`, except on PS4, where it is `gnu99`.

Supported values for the C++ language are:

`c++98`
`c++03`

ISO C++ 1998 with amendments

`gnu++98`
`gnu++03`

ISO C++ 1998 with amendments and GNU extensions

`c++11`

ISO C++ 2011 with amendments

`gnu++11`

ISO C++ 2011 with amendments and GNU extensions

`c++14`

ISO C++ 2014 with amendments

`gnu++14`

ISO C++ 2014 with amendments and GNU extensions

`c++17`

ISO C++ 2017 with amendments

`gnu++17`

ISO C++ 2017 with amendments and GNU extensions

`c++2a`

Working draft for ISO C++ 2020

`gnu++2a`

Working draft for ISO C++ 2020 with GNU extensions

The default C++ language standard is `gnu++14`.

Supported values for the OpenCL language are:

`cl1.0`

OpenCL 1.0

`cl1.1`

OpenCL 1.1

`cl1.2`

OpenCL 1.2

`cl2.0`

OpenCL 2.0

The default OpenCL language standard is `cl1.0`.

Supported values for the CUDA language are:

`cuda`

NVIDIA CUDA(tm)

-stdlib=<library>
Specify the C++ standard library to use; supported options are libstdc++ and libc++. If not specified, platform default will be used.

-rtlib=<library>
Specify the compiler runtime library to use; supported options are libgcc and compiler-rt. If not specified, platform default will be used.

-ansi
Same as -std=c89.

-ObjC, -ObjC++
Treat source input files as Objective-C and Object-C++ inputs respectively.

-trigraphs
Enable trigraphs.

-ffreestanding
Indicate that the file should be compiled for a freestanding, not a hosted, environment. Note that it is assumed that a freestanding environment will additionally provide *memcpy*, *memmove*, *memset* and *memcmp* implementations, as these are needed for efficient codegen for many programs.

-fno-builtin
Disable special handling and optimizations of well-known library functions, like **`strlen()`** and **`malloc()`**.

-fno-builtin-<function>
Disable special handling and optimizations for the specific library function. For example, `-fno-builtin-strlen` removes any special handling for the **`strlen()`** library function.

-fno-builtin-std-<function>
Disable special handling and optimizations for the specific C++ standard library function in namespace `std`. For example, `-fno-builtin-std-move_if_noexcept` removes any special handling for the **`std::move_if_noexcept()`** library function.
For C standard library functions that the C++ standard library also provides in namespace `std`, use **`-fno-builtin-<function>`** instead.

-fmath-errno
Indicate that math functions should be treated as updating **`errno`**.

-fpascal-strings
Enable support for Pascal-style strings with “\pfoo”.

-fms-extensions
Enable support for Microsoft extensions.

-fmsc-version=
Set _MSC_VER. Defaults to 1300 on Windows. Not set otherwise.

-fborland-extensions
Enable support for Borland extensions.

-fwritable-strings
Make all string literals default to writable. This disables uniquing of strings and other optimizations.

-flax-vector-conversions, -flax-vector-conversions=<kind>,
-fno-lax-vector-conversions
Allow loose type checking rules for implicit vector conversions. Possible values of <kind>:

- `none`: allow no implicit conversions between vectors
- `integer`: allow implicit bitcasts between integer vectors of the same overall bit-width
- `all`: allow implicit bitcasts between any vectors of the same overall bit-width

<kind> defaults to `integer` if unspecified.

-fblocks
Enable the “Blocks” language feature.

-fobjc-abi-version=version
Select the Objective-C ABI version to use. Available versions are 1 (legacy “fragile” ABI), 2 (non-fragile ABI 1), and 3 (non-fragile ABI 2).

-fobjc-nonfragile-abi-version=<version>
Select the Objective-C non-fragile ABI version to use by default. This will only be used as the Objective-C ABI when the non-fragile ABI is enabled (either via **-fobjc-nonfragile-abi**, or because it is the platform default).

-fobjc-nonfragile-abi, -fno-objc-nonfragile-abi
Enable use of the Objective-C non-fragile ABI. On platforms for which this is the default ABI, it can be disabled with **-fno-objc-nonfragile-abi**.

Target Selection Options

Clang fully supports cross compilation as an inherent part of its design. Depending on how your version of Clang is configured, it may have support for a number of cross compilers, or may only support a native target.

-arch <architecture>
Specify the architecture to build for (Mac OS X specific).

-target <architecture>
Specify the architecture to build for (all platforms).

-mmacosx-version-min=<version>
When building for macOS, specify the minimum version supported by your application.

-miphoneos-version-min
When building for iPhone OS, specify the minimum version supported by your application.

--print-supported-cpus
Print out a list of supported processors for the given target (specified through `--target=<architecture>` or **-arch** <architecture>). If no target is specified, the system default target will be used.

-mcpu=?, **-mtune**=?
Acts as an alias for **--print-supported-cpus**.

-march=<cpu>
Specify that Clang should generate code for a specific processor family member and later. For example, if you specify -march=i486, the compiler is allowed to generate instructions that are valid on i486 and later processors, but which may not exist on earlier ones.

Code Generation Options

-O0, -O1, -O2, -O3, -Ofast, -Os, -Oz, -Og, -O, -O4
Specify which optimization level to use:

-O0 Means “no optimization”: this level compiles the fastest and generates the most debuggable code.

-O1 Somewhere between **-O0** and **-O2**.

-O2 Moderate level of optimization which enables most optimizations.

-O3 Like **-O2**, except that it enables optimizations that take longer to perform or that may generate larger code (in an attempt to make the program run faster).

-Ofast Enables all the optimizations from **-O3** along with other aggressive optimizations that may violate strict compliance with language standards.

-Os Like **-O2** with extra optimizations to reduce code size.

-Oz Like **-Os** (and thus **-O2**), but reduces code size further.

-Og Like **-O1**. In future versions, this option might disable different optimizations in order to improve debuggability.

-O Equivalent to **-O1**.

-O4 and higher

Currently equivalent to **-O3**

-g, -gline-tables-only, -gmodules
Control debug information output. Note that Clang debug information works best at **-O0**. When more than one option starting with *-g* is specified, the last one wins:

-g Generate debug information.

-gline-tables-only Generate only line table debug information. This allows for symbolicated backtraces with inlining information, but does not include any information about variables, their locations or types.

-gmodules Generate debug information that contains external references to types defined in Clang modules or precompiled headers instead of emitting redundant debug type information into every object file. This option transparently switches the Clang module format to object file containers that hold the Clang module together with the debug information. When compiling a program that uses Clang modules or precompiled headers, this option produces complete debug information with faster compile times and much smaller object files.

This option should not be used when building static libraries for distribution to other machines because the debug info will contain references to the module cache on the machine the object files in the library were built on.

-fstandalone-debug `-fno-standalone-debug`
Clang supports a number of optimizations to reduce the size of debug information in the binary. They work based on the assumption that the debug type information can be spread out over multiple compilation units. For instance, Clang will not emit type definitions for types that are not needed by a module and could be replaced with a forward declaration. Further, Clang will only emit type info for a dynamic C++ class in the module that contains the vtable for the class.
The **-fstandalone-debug** option turns off these optimizations. This is useful when working with 3rd-party libraries that don’t come with debug information. This is the default on Darwin. Note that Clang will never emit type information for types that are not referenced at all by the program.

-feliminate-unused-debug-types
By default, Clang does not emit type information for types that are defined but not used in a program. To retain the debug info for these unused types, the negation **-fno-eliminate-unused-debug-types** can be used.

-fexceptions
Enable generation of unwind information. This allows exceptions to be thrown through Clang compiled stack frames. This is on by default in x86-64.

-ftrapv
Generate code to catch integer overflow errors. Signed integer overflow is undefined in C. With this flag, extra code is generated to detect this and abort when it happens.

-fvisibility
This flag sets the default visibility level.

-fcommon, **-fno-common**
This flag specifies that variables without initializers get common linkage. It can be disabled with **-fno-common**.

-ftls-model=<model>
Set the default thread-local storage (TLS) model to use for thread-local variables. Valid values are: “global-dynamic”, “local-dynamic”, “initial-exec” and “local-exec”. The default is “global-dynamic”. The default model can be overridden with the tls_model attribute. The compiler will try to choose a more efficient model if possible.

-flto, **-flto**=full, **-flto**=thin, **-emit-llvm**
Generate output files in LLVM formats, suitable for link time optimization. When used with **-S** this generates LLVM intermediate language assembly files, otherwise this generates LLVM bitcode format object files (which may be passed to the linker depending on the stage selection options).
The default for **-flto** is “full”, in which the LLVM bitcode is suitable for monolithic Link Time Optimization (LTO), where the linker merges all such modules into a single combined module for optimization. With “thin”, ThinLTO compilation is invoked instead.

> **Note**
>
> On Darwin, when using **-flto** along with **-g** and compiling and linking in separate steps, you also need to pass `-Wl,-object_path_lto,<lto-filename>.o` at the linking step to instruct the ld64 linker not to delete the temporary object file generated during Link Time Optimization (this flag is automatically passed to the linker by Clang if compilation and linking are done in a single step). This allows debugging the executable as well as generating the `.dSYM` bundle using dsymutil(1).

Driver Options

-###
Print (but do not run) the commands to run for this compilation.

--help
Display available options.

-Qunused-arguments
Do not emit any warnings for unused driver arguments.

-Wa,<args>
Pass the comma separated arguments in args to the assembler.

-Wl,<args>
Pass the comma separated arguments in args to the linker.

-Wp,<args>
Pass the comma separated arguments in args to the preprocessor.

-Xanalyzer <arg>
Pass arg to the static analyzer.

-Xassembler <arg>
Pass arg to the assembler.

-Xlinker <arg>
Pass arg to the linker.

-Xpreprocessor <arg>
Pass arg to the preprocessor.

-o <file>
Write output to file.

-print-file-name=<file>
Print the full library path of file.

-print-libgcc-file-name
Print the library path for the currently used compiler runtime library (“libgcc.a” or “libclang_rt.builtins.*.a”).

-print-prog-name=<name>
Print the full program path of name.

-print-search-dirs
Print the paths used for finding libraries and programs.

-save-temps
Save intermediate compilation results.

-save-stats, **-save-stats**=cwd, **-save-stats**=obj
Save internal code generation (LLVM) statistics to a file in the current directory (**-save-stats**/”-save-stats=cwd”) or the directory of the output file (“-save-state=obj”).

-integrated-as, **-no-integrated-as**
Used to enable and disable, respectively, the use of the integrated assembler. Whether the integrated assembler is on by default is target dependent.

-time
Time individual commands.

-ftime-report
Print timing summary of each stage of compilation.

-v
Show commands to run and use verbose output.

Diagnostics Options

-fshow-column, **-fshow-source-location**, **-fcaret-diagnostics**, **-fdiagnostics-fixit-info**, **-fdiagnostics-parseable-fixits**, **-fdiagnostics-print-source-range-info**, **-fprint-source-range-info**, **-fdiagnostics-show-option**, **-fmessage-length**
These options control how Clang prints out information about diagnostics (errors and warnings). Please see the Clang User’s Manual for more information.

Preprocessor Options

-D<macroname>=<value>
Adds an implicit #define into the predefines buffer which is read before the source file is preprocessed.

-U<macroname>
Adds an implicit #undef into the predefines buffer which is read before the source file is preprocessed.

-include <filename>
Adds an implicit #include into the predefines buffer which is read before the source file is preprocessed.

-I<directory>
Add the specified directory to the search path for include files.

-F<directory>
Add the specified directory to the search path for framework include files.

-nostdinc
Do not search the standard system directories or compiler builtin directories for include files.

-nostdlibinc
Do not search the standard system directories for include files, but do search compiler builtin include directories.

-nobuiltininc
Do not search clang’s builtin directory for include files.

ENVIRONMENT

TMPDIR, **TEMP**, **TMP**
These environment variables are checked, in order, for the location to write temporary files used during the compilation process.

CPATH

If this environment variable is present, it is treated as a delimited list of paths to be added to the default system include path list. The delimiter is the platform dependent delimiter, as used in the PATH environment variable. Empty components in the environment variable are ignored.

C_INCLUDE_PATH, OBJC_INCLUDE_PATH, CPLUS_INCLUDE_PATH, OBJCPLUS_INCLUDE_PATH
These environment variables specify additional paths, as for **CPATH**, which are only used when processing the appropriate language.

MACOSX_DEPLOYMENT_TARGET
If **-mmacosx-version-min** is unspecified, the default deployment target is read from this environment variable. This option only affects Darwin targets.

BUGS

To report bugs, please visit <https://github.com/llvm/llvm-project/issues/>. Most bug reports should include preprocessed source files (use the **-E** option) and the full output of the compiler, along with information to reproduce.

SEE ALSO

as(1), ld(1)

diagtool - clang diagnostics tool

SYNOPSIS

diagtool *command* [*args*]

DESCRIPTION

diagtool is a combination of four tools for dealing with diagnostics in **clang**.

SUBCOMMANDS

diagtool is separated into several subcommands each tailored to a different purpose. A brief summary of each command follows, with more detail in the sections that follow.

- find-diagnostic-id - Print the id of the given diagnostic.
- list-warnings - List warnings and their corresponding flags.
- show-enabled - Show which warnings are enabled for a given command line.
- tree - Show warning flags in a tree view.

find-diagnostic-id

diagtool find-diagnostic-id *diagnostic-name*

list-warnings

diagtool list-warnings

show-enabled

diagtool show-enabled [*options*] *filename …*

tree

diagtool tree [*diagnostic-group*]

Frequently Asked Questions (FAQ)

Driver 756

I run `clang -cc1 ...` and get weird errors about missing headers 756

I get errors about some headers being missing (`stddef.h`, `stdarg.h`) 756

Driver

I run `clang -cc1 ...` and get weird errors about missing headers

Given this source file:

```
#include <stdio.h>

int main() {
  printf("Hello world\n");
}
```

If you run:

```
$ clang -cc1 hello.c
hello.c:1:10: fatal error: 'stdio.h' file not found
#include <stdio.h>
         ^
1 error generated.
```

`clang -cc1` is the frontend, `clang` is the driver. The driver invokes the frontend with options appropriate for your system. To see these options, run:

```
$ clang -### -c hello.c
```

Some clang command line options are driver-only options, some are frontend-only options. Frontend-only options are intended to be used only by clang developers. Users should not run `clang -cc1` directly, because `-cc1` options are not guaranteed to be stable.

If you want to use a frontend-only option ("a `-cc1` option"), for example `-ast-dump`, then you need to take the `clang -cc1` line generated by the driver and add the option you need. Alternatively, you can run `clang -Xclang <option> ...` to force the driver pass `<option>` to `clang -cc1`.

I get errors about some headers being missing (`stddef.h`, `stdarg.h`)

Some header files (`stddef.h`, `stdarg.h`, and others) are shipped with Clang — these are called builtin includes. Clang searches for them in a directory relative to the location of the `clang` binary. If you moved the `clang` binary, you need to move the builtin headers, too.

More information can be found in the Builtin includes section.

Using Clang as a Library

Choosing the Right Interface for Your Application

Clang provides infrastructure to write tools that need syntactic and semantic information about a program. This document will give a short introduction of the different ways to write clang tools, and their pros and cons.

LibClang

LibClang is a stable high level C interface to clang. When in doubt LibClang is probably the interface you want to use. Consider the other interfaces only when you have a good reason not to use LibClang.

Canonical examples of when to use LibClang:

- Xcode
- Clang Python Bindings

Use LibClang when you…:

- want to interface with clang from other languages than C++
- need a stable interface that takes care to be backwards compatible
- want powerful high-level abstractions, like iterating through an AST with a cursor, and don't want to learn all the nitty gritty details of Clang's AST.

Do not use LibClang when you…:

- want full control over the Clang AST

Clang Plugins

Clang Plugins allow you to run additional actions on the AST as part of a compilation. Plugins are dynamic libraries that are loaded at runtime by the compiler, and they're easy to integrate into your build environment.

Canonical examples of when to use Clang Plugins:

- special lint-style warnings or errors for your project
- creating additional build artifacts from a single compile step

Use Clang Plugins when you…:

- need your tool to rerun if any of the dependencies change
- want your tool to make or break a build
- need full control over the Clang AST

Do not use Clang Plugins when you…:

- want to run tools outside of your build environment
- want full control on how Clang is set up, including mapping of in-memory virtual files
- need to run over a specific subset of files in your project which is not necessarily related to any changes which would trigger rebuilds

LibTooling

LibTooling is a C++ interface aimed at writing standalone tools, as well as integrating into services that run clang tools. Canonical examples of when to use LibTooling:

- a simple syntax checker
- refactoring tools

Use LibTooling when you…:

- want to run tools over a single file, or a specific subset of files, independently of the build system
- want full control over the Clang AST
- want to share code with Clang Plugins

Do not use LibTooling when you…:

- want to run as part of the build triggered by dependency changes
- want a stable interface so you don't need to change your code when the AST API changes
- want high level abstractions like cursors and code completion out of the box
- do not want to write your tools in C++

Clang tools are a collection of specific developer tools built on top of the LibTooling infrastructure as part of the Clang project. They are targeted at automating and improving core development activities of C/C++ developers.

Examples of tools we are building or planning as part of the Clang project:

- Syntax checking (**clang-check**)
- Automatic fixing of compile errors (**clang-fixit**)
- Automatic code formatting (**clang-format**)
- Migration tools for new features in new language standards
- Core refactoring tools

External Clang Examples

Introduction

This page provides some examples of the kinds of things that people have done with Clang that might serve as useful guides (or starting points) from which to develop your own tools. They may be helpful even for something as banal (but necessary) as how to set up your build to integrate Clang.

Clang's library-based design is deliberately aimed at facilitating use by external projects, and we are always interested in improving Clang to better serve our external users. Some typical categories of applications where Clang is used are:

- Static analysis.
- Documentation/cross-reference generation.

If you know of (or wrote!) a tool or project using Clang, please send an email to Clang's development discussion mailing list to have it added. (or if you are already a Clang contributor, feel free to directly commit additions). Since the primary purpose of this page is to provide examples that can help developers, generally they must have code available.

List of projects and tools

https://github.com/Andersbakken/rtags/

"RTags is a client/server application that indexes c/c++ code and keeps a persistent in-memory database of references, symbolnames, completions etc."

https://rprichard.github.com/sourceweb/

"A C/C++ source code indexer and navigator"

https://github.com/etaoins/qconnectlint

"qconnectlint is a Clang tool for statically verifying the consistency of signal and slot connections made with Qt's `QObject::connect`."

https://github.com/woboq/woboq_codebrowser

"The Woboq Code Browser is a web-based code browser for C/C++ projects. Check out https://code.woboq.org/ for an example!"

https://github.com/mozilla/dxr

"DXR is a source code cross-reference tool that uses static analysis data collected by instrumented compilers."

https://github.com/eschulte/clang-mutate

"This tool performs a number of operations on C-language source files."

https://github.com/gmarpons/Crisp

"A coding rule validation add-on for LLVM/clang. Crisp rules are written in Prolog. A high-level declarative DSL to easily write new rules is under development. It will be called CRISP, an acronym for *Coding Rules in Sugared Prolog*."

https://github.com/drothlis/clang-ctags

"Generate tag file for C++ source code."

https://github.com/exclipy/clang_indexer

"This is an indexer for C and C++ based on the libclang library."

https://github.com/holtgrewe/linty

"Linty - C/C++ Style Checking with Python & libclang."

https://github.com/axw/cmonster

"cmonster is a Python wrapper for the Clang C++ parser."

https://github.com/rizsotto/Constantine

"Constantine is a toy project to learn how to write clang plugin. Implements pseudo const analysis. Generates warnings about variables, which were declared without const qualifier."

https://github.com/jessevdk/cldoc

"cldoc is a Clang based documentation generator for C and C++. cldoc tries to solve the issue of writing C/C++ software documentation with a modern, non-intrusive and robust approach."

https://github.com/AlexDenisov/ToyClangPlugin

"The simplest Clang plugin implementing a semantic check for Objective-C. This example shows how to use the `DiagnosticsEngine` (emit warnings, errors, fixit hints). See also http://l.rw.rw/clang_plugin for step-by-step instructions."

https://phabricator.kde.org/source/clazy

"clazy is a compiler plugin which allows clang to understand Qt semantics. You get more than 50 Qt related compiler warnings, ranging from unneeded memory allocations to misusage of API, including fix-its for automatic refactoring."

https://gerrit.libreoffice.org/gitweb?p=core.git;a=blob_plain;f=compilerplugins/README;hb=HEAD

"LibreOffice uses a Clang plugin infrastructure to check during the build various things, some more, some less specific to the LibreOffice source code. There are currently around 50 such checkers, from flagging C-style casts and uses of reserved identifiers to ensuring that code adheres to lifecycle protocols for certain LibreOffice-specific classes. They may serve as examples for writing RecursiveASTVisitor-based plugins."

Introduction to the Clang AST

This document gives a gentle introduction to the mysteries of the Clang AST. It is targeted at developers who either want to contribute to Clang, or use tools that work based on Clang's AST, like the AST matchers.

Slides

Introduction

Clang's AST is different from ASTs produced by some other compilers in that it closely resembles both the written C++ code and the C++ standard. For example, parenthesis expressions and compile time constants are available in an unreduced form in the AST. This makes Clang's AST a good fit for refactoring tools.

Documentation for all Clang AST nodes is available via the generated Doxygen. The doxygen online documentation is also indexed by your favorite search engine, which will make a search for clang and the AST node's class name usually turn up the doxygen of the class you're looking for (for example, search for: clang ParenExpr).

Examining the AST

A good way to familiarize yourself with the Clang AST is to actually look at it on some simple example code. Clang has a builtin AST-dump mode, which can be enabled with the flag `-ast-dump`.

Let's look at a simple example AST:

```
$ cat test.cc
int f(int x) {
  int result = (x / 42);
  return result;
}

# Clang by default is a frontend for many tools; -Xclang is used to pass
# options directly to the C++ frontend.
$ clang -Xclang -ast-dump -fsyntax-only test.cc
TranslationUnitDecl 0x5aea0d0 <<invalid sloc>>
... cutting out internal declarations of clang ...
`-FunctionDecl 0x5aeab50 <test.cc:1:1, line:4:1> f 'int (int)'
```

```
|-ParmVarDecl 0x5aeaa90 <line:1:7, col:11> x 'int'
`-CompoundStmt 0x5aead88 <col:14, line:4:1>
  |-DeclStmt 0x5aead10 <line:2:3, col:24>
  | `-VarDecl 0x5aeac10 <col:3, col:23> result 'int'
  |   `-ParenExpr 0x5aeacf0 <col:16, col:23> 'int'
  |     `-BinaryOperator 0x5aeacc8 <col:17, col:21> 'int' '/'
  |       |-ImplicitCastExpr 0x5aeacb0 <col:17> 'int' <LValueToRValue>
  |       | `-DeclRefExpr 0x5aeac68 <col:17> 'int' lvalue ParmVar 0x5aeaa90 'x' 'int'
  |       `-IntegerLiteral 0x5aeac90 <col:21> 'int' 42
  `-ReturnStmt 0x5aead68 <line:3:3, col:10>
    `-ImplicitCastExpr 0x5aead50 <col:10> 'int' <LValueToRValue>
      `-DeclRefExpr 0x5aead28 <col:10> 'int' lvalue Var 0x5aeac10 'result' 'int'
```

The toplevel declaration in a translation unit is always the translation unit declaration. In this example, our first user written declaration is the function declaration of “f”. The body of “f” is a compound statement, whose child nodes are a declaration statement that declares our result variable, and the return statement.

AST Context

All information about the AST for a translation unit is bundled up in the class ASTContext. It allows traversal of the whole translation unit starting from getTranslationUnitDecl, or to access Clang’s table of identifiers for the parsed translation unit.

AST Nodes

Clang’s AST nodes are modeled on a class hierarchy that does not have a common ancestor. Instead, there are multiple larger hierarchies for basic node types like Decl and Stmt. Many important AST nodes derive from Type, Decl, DeclContext or Stmt, with some classes deriving from both Decl and DeclContext.

There are also a multitude of nodes in the AST that are not part of a larger hierarchy, and are only reachable from specific other nodes, like CXXBaseSpecifier.

Thus, to traverse the full AST, one starts from the TranslationUnitDecl and then recursively traverses everything that can be reached from that node - this information has to be encoded for each specific node type. This algorithm is encoded in the RecursiveASTVisitor. See the RecursiveASTVisitor tutorial.

The two most basic nodes in the Clang AST are statements (Stmt) and declarations (Decl). Note that expressions (Expr) are also statements in Clang’s AST.

LibTooling

LibTooling is a library to support writing standalone tools based on Clang. This document will provide a basic walkthrough of how to write a tool using LibTooling.

For the information on how to setup Clang Tooling for LLVM see How To Setup Clang Tooling For LLVM

Introduction

Tools built with LibTooling, like Clang Plugins, run `FrontendActions` over code.

In this tutorial, we’ll demonstrate the different ways of running Clang’s `SyntaxOnlyAction`, which runs a quick syntax check, over a bunch of code.

Parsing a code snippet in memory

If you ever wanted to run a `FrontendAction` over some sample code, for example to unit test parts of the Clang AST, `runToolOnCode` is what you looked for. Let me give you an example:

```
#include "clang/Tooling/Tooling.h"

TEST(runToolOnCode, CanSyntaxCheckCode) {
  // runToolOnCode returns whether the action was correctly run over the
```

```
  // given code.
  EXPECT_TRUE(runToolOnCode(std::make_unique<clang::SyntaxOnlyAction>(), "class X {};"));
}
```

Writing a standalone tool

Once you unit tested your `FrontendAction` to the point where it cannot possibly break, it's time to create a standalone tool. For a standalone tool to run clang, it first needs to figure out what command line arguments to use for a specified file. To that end we create a `CompilationDatabase`. There are different ways to create a compilation database, and we need to support all of them depending on command-line options. There's the `CommonOptionsParser` class that takes the responsibility to parse command-line parameters related to compilation databases and inputs, so that all tools share the implementation.

Parsing common tools options

`CompilationDatabase` can be read from a build directory or the command line. Using `CommonOptionsParser` allows for explicit specification of a compile command line, specification of build path using the `-p` command-line option, and automatic location of the compilation database using source files paths.

```
#include "clang/Tooling/CommonOptionsParser.h"
#include "llvm/Support/CommandLine.h"

using namespace clang::tooling;

// Apply a custom category to all command-line options so that they are the
// only ones displayed.
static llvm::cl::OptionCategory MyToolCategory("my-tool options");

int main(int argc, const char **argv) {
  // CommonOptionsParser constructor will parse arguments and create a
  // CompilationDatabase.  In case of error it will terminate the program.
  CommonOptionsParser OptionsParser(argc, argv, MyToolCategory);

  // Use OptionsParser.getCompilations() and OptionsParser.getSourcePathList()
  // to retrieve CompilationDatabase and the list of input file paths.
}
```

Creating and running a ClangTool

Once we have a `CompilationDatabase`, we can create a `ClangTool` and run our `FrontendAction` over some code. For example, to run the `SyntaxOnlyAction` over the files "a.cc" and "b.cc" one would write:

```
// A clang tool can run over a number of sources in the same process...
std::vector<std::string> Sources;
Sources.push_back("a.cc");
Sources.push_back("b.cc");

// We hand the CompilationDatabase we created and the sources to run over into
// the tool constructor.
ClangTool Tool(OptionsParser.getCompilations(), Sources);

// The ClangTool needs a new FrontendAction for each translation unit we run
// on.  Thus, it takes a FrontendActionFactory as parameter.  To create a
// FrontendActionFactory from a given FrontendAction type, we call
// newFrontendActionFactory<clang::SyntaxOnlyAction>().
int result = Tool.run(newFrontendActionFactory<clang::SyntaxOnlyAction>().get());
```

Putting it together — the first tool

Now we combine the two previous steps into our first real tool. A more advanced version of this example tool is also checked into the clang tree at `tools/clang-check/ClangCheck.cpp`.

```
// Declares clang::SyntaxOnlyAction.
#include "clang/Frontend/FrontendActions.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
// Declares llvm::cl::extrahelp.
#include "llvm/Support/CommandLine.h"

using namespace clang::tooling;
using namespace llvm;

// Apply a custom category to all command-line options so that they are the
// only ones displayed.
static cl::OptionCategory MyToolCategory("my-tool options");

// CommonOptionsParser declares HelpMessage with a description of the common
// command-line options related to the compilation database and input files.
// It's nice to have this help message in all tools.
static cl::extrahelp CommonHelp(CommonOptionsParser::HelpMessage);

// A help message for this specific tool can be added afterwards.
static cl::extrahelp MoreHelp("\nMore help text...\n");

int main(int argc, const char **argv) {
  CommonOptionsParser OptionsParser(argc, argv, MyToolCategory);
  ClangTool Tool(OptionsParser.getCompilations(),
                 OptionsParser.getSourcePathList());
  return Tool.run(newFrontendActionFactory<clang::SyntaxOnlyAction>().get());
}
```

Running the tool on some code

When you check out and build clang, clang-check is already built and available to you in bin/clang-check inside your build directory.

You can run clang-check on a file in the llvm repository by specifying all the needed parameters after a "`--`" separator:

```
$ cd /path/to/source/llvm
$ export BD=/path/to/build/llvm
$ $BD/bin/clang-check tools/clang/tools/clang-check/ClangCheck.cpp -- \
      clang++ -D__STDC_CONSTANT_MACROS -D__STDC_LIMIT_MACROS \
      -Itools/clang/include -I$BD/include -Iinclude \
      -Itools/clang/lib/Headers -c
```

As an alternative, you can also configure cmake to output a compile command database into its build directory:

```
# Alternatively to calling cmake, use ccmake, toggle to advanced mode and
# set the parameter CMAKE_EXPORT_COMPILE_COMMANDS from the UI.
$ cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON .
```

This creates a file called `compile_commands.json` in the build directory. Now you can run **clang-check** over files in the project by specifying the build path as first argument and some source files as further positional arguments:

```
$ cd /path/to/source/llvm
$ export BD=/path/to/build/llvm
$ $BD/bin/clang-check -p $BD tools/clang/tools/clang-check/ClangCheck.cpp
```

Builtin includes

Clang tools need their builtin headers and search for them the same way Clang does. Thus, the default location to look for builtin headers is in a path `$(dirname /path/to/tool)/../lib/clang/3.3/include` relative to the tool binary. This works out-of-the-box for tools running from llvm's toplevel binary directory after building clang-resource-headers, or if the tool is running from the binary directory of a clang install next to the clang binary.

Tips: if your tool fails to find `stddef.h` or similar headers, call the tool with `-v` and look at the search paths it looks through.

Linking

For a list of libraries to link, look at one of the tools' CMake files (for example clang-check/CMakeList.txt).

LibFormat

LibFormat is a library that implements automatic source code formatting based on Clang. This documents describes the LibFormat interface and design as well as some basic style discussions.

If you just want to use *clang-format* as a tool or integrated into an editor, checkout ClangFormat.

Design

FIXME: Write up design.

Interface

The core routine of LibFormat is `reformat()`:

```
tooling::Replacements reformat(const FormatStyle &Style, Lexer &Lex,
                               SourceManager &SourceMgr,
                               std::vector<CharSourceRange> Ranges);
```

This reads a token stream out of the lexer `Lex` and reformats all the code ranges in `Ranges`. The `FormatStyle` controls basic decisions made during formatting. A list of options can be found under Style Options.

The style options are described in Clang-Format Style Options.

Style Options

The style options describe specific formatting options that can be used in order to make *ClangFormat* comply with different style guides. Currently, several style guides are hard-coded:

```
/// Returns a format style complying with the LLVM coding standards:
/// https://llvm.org/docs/CodingStandards.html.
FormatStyle getLLVMStyle();

/// Returns a format style complying with Google's C++ style guide:
/// http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml.
FormatStyle getGoogleStyle();

/// Returns a format style complying with Chromium's style guide:
/// https://chromium.googlesource.com/chromium/src/+/refs/heads/main/styleguide/styleguide.md
FormatStyle getChromiumStyle();

/// Returns a format style complying with the GNU coding standards:
/// https://www.gnu.org/prep/standards/standards.html
FormatStyle getGNUStyle();

/// Returns a format style complying with Mozilla's style guide
/// https://firefox-source-docs.mozilla.org/code-quality/coding-style/index.html
FormatStyle getMozillaStyle();
```

```
/// Returns a format style complying with Webkit's style guide:
/// https://webkit.org/code-style-guidelines/
FormatStyle getWebkitStyle();

/// Returns a format style complying with Microsoft's style guide:
/// https://docs.microsoft.com/en-us/visualstudio/ide/editorconfig-code-style-settings-reference
FormatStyle getMicrosoftStyle();
```

These options are also exposed in the standalone tools through the *-style* option.

In the future, we plan on making this configurable.

Clang Plugins

Clang Plugins make it possible to run extra user defined actions during a compilation. This document will provide a basic walkthrough of how to write and run a Clang Plugin.

Introduction

Clang Plugins run FrontendActions over code. See the FrontendAction tutorial on how to write a `FrontendAction` using the `RecursiveASTVisitor`. In this tutorial, we'll demonstrate how to write a simple clang plugin.

Writing a `PluginASTAction`

The main difference from writing normal `FrontendActions` is that you can handle plugin command line options. The `PluginASTAction` base class declares a `ParseArgs` method which you have to implement in your plugin.

```
bool ParseArgs(const CompilerInstance &CI,
               const std::vector<std::string>& args) {
  for (unsigned i = 0, e = args.size(); i != e; ++i) {
    if (args[i] == "-some-arg") {
      // Handle the command line argument.
    }
  }
  return true;
}
```

Registering a plugin

A plugin is loaded from a dynamic library at runtime by the compiler. To register a plugin in a library, use `FrontendPluginRegistry::Add<>`:

```
static FrontendPluginRegistry::Add<MyPlugin> X("my-plugin-name", "my plugin description");
```

Defining pragmas

Plugins can also define pragmas by declaring a `PragmaHandler` and registering it using `PragmaHandlerRegistry::Add<>`:

```
// Define a pragma handler for #pragma example_pragma
class ExamplePragmaHandler : public PragmaHandler {
public:
  ExamplePragmaHandler() : PragmaHandler("example_pragma") { }
  void HandlePragma(Preprocessor &PP, PragmaIntroducer Introducer,
                    Token &PragmaTok) {
    // Handle the pragma
  }
};

static PragmaHandlerRegistry::Add<ExamplePragmaHandler> Y("example_pragma","example pragma description");
```

Defining attributes

Plugins can define attributes by declaring a `ParsedAttrInfo` and registering it using `ParsedAttrInfoRegister::Add<>`:

```
class ExampleAttrInfo : public ParsedAttrInfo {
public:
  ExampleAttrInfo() {
    Spellings.push_back({ParsedAttr::AS_GNU,"example"});
  }
  AttrHandling handleDeclAttribute(Sema &S, Decl *D,
                                   const ParsedAttr &Attr) const override {
    // Handle the attribute
    return AttributeApplied;
  }
};

static ParsedAttrInfoRegistry::Add<ExampleAttrInfo> Z("example_attr","example attribute description");
```

The members of `ParsedAttrInfo` that a plugin attribute must define are:

- `Spellings`, which must be populated with every Spelling of the attribute, each of which consists of an attribute syntax and how the attribute name is spelled for that syntax. If the syntax allows a scope then the spelling must be “scope::attr” if a scope is present or “::attr” if not.
- `handleDeclAttribute`, which is the function that applies the attribute to a declaration. It is responsible for checking that the attribute’s arguments are valid, and typically applies the attribute by adding an `Attr` to the `Decl`. It returns either `AttributeApplied`, to indicate that the attribute was successfully applied, or `AttributeNotApplied` if it wasn’t.

The members of `ParsedAttrInfo` that may need to be defined, depending on the attribute, are:

- `NumArgs` and `OptArgs`, which set the number of required and optional arguments to the attribute.
- `diagAppertainsToDecl`, which checks if the attribute has been used on the right kind of declaration and issues a diagnostic if not.
- `diagLangOpts`, which checks if the attribute is permitted for the current language mode and issues a diagnostic if not.
- `existsInTarget`, which checks if the attribute is permitted for the given target.

To see a working example of an attribute plugin, see the Attribute.cpp example.

Putting it all together

Let’s look at an example plugin that prints top-level function names. This example is checked into the clang repository; please take a look at the latest version of PrintFunctionNames.cpp.

Running the plugin

Using the compiler driver

The Clang driver accepts the *-fplugin* option to load a plugin. Clang plugins can receive arguments from the compiler driver command line via the *fplugin-arg-<plugin name>-<argument>* option. Using this method, the plugin name cannot contain dashes itself, but the argument passed to the plugin can.

```
$ export BD=/path/to/build/directory
$ make -C $BD CallSuperAttr
$ clang++ -fplugin=$BD/lib/CallSuperAttr.so \
          -fplugin-arg-call_super_plugin-help \
          test.cpp
```

If your plugin name contains dashes, either rename the plugin or used the cc1 command line options listed below.

Using the cc1 command line

To run a plugin, the dynamic library containing the plugin registry must be loaded via the *-load* command line option. This will load all plugins that are registered, and you can select the plugins to run by specifying the *-plugin* option. Additional parameters for the plugins can be passed with *-plugin-arg-<plugin-name>*.

Note that those options must reach clang's cc1 process. There are two ways to do so:

- Directly call the parsing process by using the *-cc1* option; this has the downside of not configuring the default header search paths, so you'll need to specify the full system path configuration on the command line.
- Use clang as usual, but prefix all arguments to the cc1 process with *-Xclang*.

For example, to run the `print-function-names` plugin over a source file in clang, first build the plugin, and then call clang with the plugin from the source tree:

```
$ export BD=/path/to/build/directory
$ (cd $BD && make PrintFunctionNames )
$ clang++ -D_GNU_SOURCE -D_DEBUG -D__STDC_CONSTANT_MACROS \
          -D__STDC_FORMAT_MACROS -D__STDC_LIMIT_MACROS -D_GNU_SOURCE \
          -I$BD/tools/clang/include -Itools/clang/include -I$BD/include -Iinclude \
          tools/clang/tools/clang-check/ClangCheck.cpp -fsyntax-only \
          -Xclang -load -Xclang $BD/lib/PrintFunctionNames.so -Xclang \
          -plugin -Xclang print-fns
```

Also see the print-function-name plugin example's README

Using the clang command line

Using *-fplugin=plugin* on the clang command line passes the plugin through as an argument to *-load* on the cc1 command line. If the plugin class implements the `getActionType` method then the plugin is run automatically. For example, to run the plugin automatically after the main AST action (i.e. the same as using *-add-plugin*):

```
// Automatically run the plugin after the main AST action
PluginASTAction::ActionType getActionType() override {
  return AddAfterMainAction;
}
```

Interaction with `-clear-ast-before-backend`

To reduce peak memory usage of the compiler, plugins are recommended to run *before* the main action, which is usually code generation. This is because having any plugins that run after the codegen action automatically turns off `-clear-ast-before-backend`. `-clear-ast-before-backend` reduces peak memory by clearing the Clang AST after generating IR and before running IR optimizations. Use `CmdlineBeforeMainAction` or `AddBeforeMainAction` as `getActionType` to run plugins while still benefitting from `-clear-ast-before-backend`. Plugins must make sure not to modify the AST, otherwise they should run after the main action.

How to write RecursiveASTVisitor based ASTFrontendActions.

Introduction

In this tutorial you will learn how to create a FrontendAction that uses a RecursiveASTVisitor to find CXXRecordDecl AST nodes with a specified name.

Creating a FrontendAction

When writing a clang based tool like a Clang Plugin or a standalone tool based on LibTooling, the common entry point is the FrontendAction. FrontendAction is an interface that allows execution of user specific actions as part of the compilation. To run tools over the AST clang provides the convenience interface ASTFrontendAction, which takes care of executing the action. The only part left is to implement the CreateASTConsumer method that returns an ASTConsumer per translation unit.

```
class FindNamedClassAction : public clang::ASTFrontendAction {
public:
  virtual std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
    clang::CompilerInstance &Compiler, llvm::StringRef InFile) {
    return std::make_unique<FindNamedClassConsumer>();
  }
};
```

Creating an ASTConsumer

ASTConsumer is an interface used to write generic actions on an AST, regardless of how the AST was produced. ASTConsumer provides many different entry points, but for our use case the only one needed is HandleTranslationUnit, which is called with the ASTContext for the translation unit.

```
class FindNamedClassConsumer : public clang::ASTConsumer {
public:
  virtual void HandleTranslationUnit(clang::ASTContext &Context) {
    // Traversing the translation unit decl via a RecursiveASTVisitor
    // will visit all nodes in the AST.
    Visitor.TraverseDecl(Context.getTranslationUnitDecl());
  }
private:
  // A RecursiveASTVisitor implementation.
  FindNamedClassVisitor Visitor;
};
```

Using the RecursiveASTVisitor

Now that everything is hooked up, the next step is to implement a RecursiveASTVisitor to extract the relevant information from the AST.

The RecursiveASTVisitor provides hooks of the form bool VisitNodeType(NodeType *) for most AST nodes; the exception are TypeLoc nodes, which are passed by-value. We only need to implement the methods for the relevant node types.

Let's start by writing a RecursiveASTVisitor that visits all CXXRecordDecl's.

```
class FindNamedClassVisitor
  : public RecursiveASTVisitor<FindNamedClassVisitor> {
public:
  bool VisitCXXRecordDecl(CXXRecordDecl *Declaration) {
    // For debugging, dumping the AST nodes will show which nodes are already
    // being visited.
    Declaration->dump();

    // The return value indicates whether we want the visitation to proceed.
    // Return false to stop the traversal of the AST.
    return true;
  }
};
```

In the methods of our RecursiveASTVisitor we can now use the full power of the Clang AST to drill through to the parts that are interesting for us. For example, to find all class declaration with a certain name, we can check for a specific qualified name:

```
bool VisitCXXRecordDecl(CXXRecordDecl *Declaration) {
  if (Declaration->getQualifiedNameAsString() == "n::m::C")
    Declaration->dump();
  return true;
}
```

Accessing the SourceManager and ASTContext

Some of the information about the AST, like source locations and global identifier information, are not stored in the AST nodes themselves, but in the ASTContext and its associated source manager. To retrieve them we need to hand the ASTContext into our RecursiveASTVisitor implementation.

The ASTContext is available from the CompilerInstance during the call to CreateASTConsumer. We can thus extract it there and hand it into our freshly created FindNamedClassConsumer:

```
virtual std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
  clang::CompilerInstance &Compiler, llvm::StringRef InFile) {
  return std::make_unique<FindNamedClassConsumer>(&Compiler.getASTContext());
}
```

Now that the ASTContext is available in the RecursiveASTVisitor, we can do more interesting things with AST nodes, like looking up their source locations:

```
bool VisitCXXRecordDecl(CXXRecordDecl *Declaration) {
  if (Declaration->getQualifiedNameAsString() == "n::m::C") {
    // getFullLoc uses the ASTContext's SourceManager to resolve the source
    // location and break it up into its line and column parts.
    FullSourceLoc FullLocation = Context->getFullLoc(Declaration->getBeginLoc());
    if (FullLocation.isValid())
      llvm::outs() << "Found declaration at "
                   << FullLocation.getSpellingLineNumber() << ":"
                   << FullLocation.getSpellingColumnNumber() << "\n";
  }
  return true;
}
```

Putting it all together

Now we can combine all of the above into a small example program:

```
#include "clang/AST/ASTConsumer.h"
#include "clang/AST/RecursiveASTVisitor.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/FrontendAction.h"
#include "clang/Tooling/Tooling.h"

using namespace clang;

class FindNamedClassVisitor
  : public RecursiveASTVisitor<FindNamedClassVisitor> {
public:
  explicit FindNamedClassVisitor(ASTContext *Context)
    : Context(Context) {}

  bool VisitCXXRecordDecl(CXXRecordDecl *Declaration) {
    if (Declaration->getQualifiedNameAsString() == "n::m::C") {
      FullSourceLoc FullLocation = Context->getFullLoc(Declaration->getBeginLoc());
      if (FullLocation.isValid())
        llvm::outs() << "Found declaration at "
                     << FullLocation.getSpellingLineNumber() << ":"
                     << FullLocation.getSpellingColumnNumber() << "\n";
    }
    return true;
  }

private:
  ASTContext *Context;
};
```

```
class FindNamedClassConsumer : public clang::ASTConsumer {
public:
  explicit FindNamedClassConsumer(ASTContext *Context)
    : Visitor(Context) {}

  virtual void HandleTranslationUnit(clang::ASTContext &Context) {
    Visitor.TraverseDecl(Context.getTranslationUnitDecl());
  }
private:
  FindNamedClassVisitor Visitor;
};

class FindNamedClassAction : public clang::ASTFrontendAction {
public:
  virtual std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
    clang::CompilerInstance &Compiler, llvm::StringRef InFile) {
    return std::make_unique<FindNamedClassConsumer>(&Compiler.getASTContext());
  }
};

int main(int argc, char **argv) {
  if (argc > 1) {
    clang::tooling::runToolOnCode(std::make_unique<FindNamedClassAction>(), argv[1]);
  }
}
```

We store this into a file called FindClassDecls.cpp and create the following CMakeLists.txt to link it:

```
set(LLVM_LINK_COMPONENTS
  Support
  )

add_clang_executable(find-class-decls FindClassDecls.cpp)

target_link_libraries(find-class-decls
  PRIVATE
  clangAST
  clangBasic
  clangFrontend
  clangSerialization
  clangTooling
  )
```

When running this tool over a small code snippet it will output all declarations of a class n::m::C it found:

```
$ ./bin/find-class-decls "namespace n { namespace m { class C {}; } }"
Found declaration at 1:29
```

Tutorial for building tools using LibTooling and LibASTMatchers

This document is intended to show how to build a useful source-to-source translation tool based on Clang's LibTooling. It is explicitly aimed at people who are new to Clang, so all you should need is a working knowledge of C++ and the command line.

In order to work on the compiler, you need some basic knowledge of the abstract syntax tree (AST). To this end, the reader is encouraged to skim the Introduction to the Clang AST

Step 0: Obtaining Clang

As Clang is part of the LLVM project, you'll need to download LLVM's source code first. Both Clang and LLVM are in the same git repository, under different directories. For further information, see the getting started guide.

```
cd ~/clang-llvm
git clone https://github.com/llvm/llvm-project.git
```

Next you need to obtain the CMake build system and Ninja build tool.

```
cd ~/clang-llvm
git clone https://github.com/martine/ninja.git
cd ninja
git checkout release
./bootstrap.py
sudo cp ninja /usr/bin/

cd ~/clang-llvm
git clone git://cmake.org/stage/cmake.git
cd cmake
git checkout next
./bootstrap
make
sudo make install
```

Okay. Now we'll build Clang!

```
cd ~/clang-llvm
mkdir build && cd build
cmake -G Ninja ../llvm -DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra" -DLLVM_BUILD_TESTS=ON  # Enable tests; default is off.
ninja
ninja check       # Test LLVM only.
ninja clang-test  # Test Clang only.
ninja install
```

And we're live.

All of the tests should pass.

Finally, we want to set Clang as its own compiler.

```
cd ~/clang-llvm/build
ccmake ../llvm
```

The second command will bring up a GUI for configuring Clang. You need to set the entry for `CMAKE_CXX_COMPILER`. Press `'t'` to turn on advanced mode. Scroll down to `CMAKE_CXX_COMPILER`, and set it to `/usr/bin/clang++`, or wherever you installed it. Press `'c'` to configure, then `'g'` to generate CMake's files.

Finally, run ninja one last time, and you're done.

Step 1: Create a ClangTool

Now that we have enough background knowledge, it's time to create the simplest productive ClangTool in existence: a syntax checker. While this already exists as `clang-check`, it's important to understand what's going on.

First, we'll need to create a new directory for our tool and tell CMake that it exists. As this is not going to be a core clang tool, it will live in the `clang-tools-extra` repository.

```
cd ~/clang-llvm
mkdir clang-tools-extra/loop-convert
echo 'add_subdirectory(loop-convert)' >> clang-tools-extra/CMakeLists.txt
vim clang-tools-extra/loop-convert/CMakeLists.txt
```

CMakeLists.txt should have the following contents:

```
set(LLVM_LINK_COMPONENTS support)

add_clang_executable(loop-convert
  LoopConvert.cpp
  )
```

```
target_link_libraries(loop-convert
  PRIVATE
  clangAST
  clangASTMatchers
  clangBasic
  clangFrontend
  clangSerialization
  clangTooling
  )
```

With that done, Ninja will be able to compile our tool. Let's give it something to compile! Put the following into `clang-tools-extra/loop-convert/LoopConvert.cpp`. A detailed explanation of why the different parts are needed can be found in the LibTooling documentation.

```
// Declares clang::SyntaxOnlyAction.
#include "clang/Frontend/FrontendActions.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
// Declares llvm::cl::extrahelp.
#include "llvm/Support/CommandLine.h"

using namespace clang::tooling;
using namespace llvm;

// Apply a custom category to all command-line options so that they are the
// only ones displayed.
static llvm::cl::OptionCategory MyToolCategory("my-tool options");

// CommonOptionsParser declares HelpMessage with a description of the common
// command-line options related to the compilation database and input files.
// It's nice to have this help message in all tools.
static cl::extrahelp CommonHelp(CommonOptionsParser::HelpMessage);

// A help message for this specific tool can be added afterwards.
static cl::extrahelp MoreHelp("\nMore help text...\n");

int main(int argc, const char **argv) {
  auto ExpectedParser = CommonOptionsParser::create(argc, argv, MyToolCategory);
  if (!ExpectedParser) {
    // Fail gracefully for unsupported options.
    llvm::errs() << ExpectedParser.takeError();
    return 1;
  }
  CommonOptionsParser& OptionsParser = ExpectedParser.get();
  ClangTool Tool(OptionsParser.getCompilations(),
                 OptionsParser.getSourcePathList());
  return Tool.run(newFrontendActionFactory<clang::SyntaxOnlyAction>().get());
}
```

And that's it! You can compile our new tool by running ninja from the `build` directory.

```
cd ~/clang-llvm/build
ninja
```

You should now be able to run the syntax checker, which is located in `~/clang-llvm/build/bin`, on any source file. Try it!

```
echo "int main() { return 0; }" > test.cpp
bin/loop-convert test.cpp --
```

Note the two dashes after we specify the source file. The additional options for the compiler are passed after the dashes rather than loading them from a compilation database - there just aren't any options needed right now.

Intermezzo: Learn AST matcher basics

Clang recently introduced the ASTMatcher library to provide a simple, powerful, and concise way to describe specific patterns in the AST. Implemented as a DSL powered by macros and templates (see ASTMatchers.h if you're curious), matchers offer the feel of algebraic data types common to functional programming languages.

For example, suppose you wanted to examine only binary operators. There is a matcher to do exactly that, conveniently named `binaryOperator`. I'll give you one guess what this matcher does:

```
binaryOperator(hasOperatorName("+"), hasLHS(integerLiteral(equals(0))))
```

Shockingly, it will match against addition expressions whose left hand side is exactly the literal 0. It will not match against other forms of 0, such as `'\0'` or `NULL`, but it will match against macros that expand to 0. The matcher will also not match against calls to the overloaded operator `'+'`, as there is a separate `operatorCallExpr` matcher to handle overloaded operators.

There are AST matchers to match all the different nodes of the AST, narrowing matchers to only match AST nodes fulfilling specific criteria, and traversal matchers to get from one kind of AST node to another. For a complete list of AST matchers, take a look at the AST Matcher References

All matcher that are nouns describe entities in the AST and can be bound, so that they can be referred to whenever a match is found. To do so, simply call the method `bind` on these matchers, e.g.:

```
variable(hasType(isInteger())).bind("intvar")
```

Step 2: Using AST matchers

Okay, on to using matchers for real. Let's start by defining a matcher which will capture all `for` statements that define a new variable initialized to zero. Let's start with matching all `for` loops:

```
forStmt()
```

Next, we want to specify that a single variable is declared in the first portion of the loop, so we can extend the matcher to

```
forStmt(hasLoopInit(declStmt(hasSingleDecl(varDecl()))))
```

Finally, we can add the condition that the variable is initialized to zero.

```
forStmt(hasLoopInit(declStmt(hasSingleDecl(varDecl(
  hasInitializer(integerLiteral(equals(0))))))))
```

It is fairly easy to read and understand the matcher definition ("match loops whose init portion declares a single variable which is initialized to the integer literal 0"), but deciding that every piece is necessary is more difficult. Note that this matcher will not match loops whose variables are initialized to `'\0'`, `0.0`, `NULL`, or any form of zero besides the integer 0.

The last step is giving the matcher a name and binding the `ForStmt` as we will want to do something with it:

```
StatementMatcher LoopMatcher =
  forStmt(hasLoopInit(declStmt(hasSingleDecl(varDecl(
    hasInitializer(integerLiteral(equals(0)))))))).bind("forLoop");
```

Once you have defined your matchers, you will need to add a little more scaffolding in order to run them. Matchers are paired with a `MatchCallback` and registered with a `MatchFinder` object, then run from a `ClangTool`. More code!

Add the following to `LoopConvert.cpp`:

```
#include "clang/ASTMatchers/ASTMatchers.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"

using namespace clang;
using namespace clang::ast_matchers;

StatementMatcher LoopMatcher =
  forStmt(hasLoopInit(declStmt(hasSingleDecl(varDecl(
    hasInitializer(integerLiteral(equals(0)))))))).bind("forLoop");
```

```
class LoopPrinter : public MatchFinder::MatchCallback {
public :
  virtual void run(const MatchFinder::MatchResult &Result) {
    if (const ForStmt *FS = Result.Nodes.getNodeAs<clang::ForStmt>("forLoop"))
      FS->dump();
  }
};
```

And change `main()` to:

```
int main(int argc, const char **argv) {
  auto ExpectedParser = CommonOptionsParser::create(argc, argv, MyToolCategory);
  if (!ExpectedParser) {
    // Fail gracefully for unsupported options.
    llvm::errs() << ExpectedParser.takeError();
    return 1;
  }
  CommonOptionsParser& OptionsParser = ExpectedParser.get();
  ClangTool Tool(OptionsParser.getCompilations(),
                 OptionsParser.getSourcePathList());

  LoopPrinter Printer;
  MatchFinder Finder;
  Finder.addMatcher(LoopMatcher, &Printer);

  return Tool.run(newFrontendActionFactory(&Finder).get());
}
```

Now, you should be able to recompile and run the code to discover for loops. Create a new file with a few examples, and test out our new handiwork:

```
cd ~/clang-llvm/llvm/llvm_build/
ninja loop-convert
vim ~/test-files/simple-loops.cc
bin/loop-convert ~/test-files/simple-loops.cc
```

Step 3.5: More Complicated Matchers

Our simple matcher is capable of discovering for loops, but we would still need to filter out many more ourselves. We can do a good portion of the remaining work with some cleverly chosen matchers, but first we need to decide exactly which properties we want to allow.

How can we characterize for loops over arrays which would be eligible for translation to range-based syntax? Range based loops over arrays of size `N` that:

- start at index `0`
- iterate consecutively
- end at index `N-1`

We already check for (1), so all we need to add is a check to the loop's condition to ensure that the loop's index variable is compared against `N` and another check to ensure that the increment step just increments this same variable. The matcher for (2) is straightforward: require a pre- or post-increment of the same variable declared in the init portion.

Unfortunately, such a matcher is impossible to write. Matchers contain no logic for comparing two arbitrary AST nodes and determining whether or not they are equal, so the best we can do is matching more than we would like to allow, and punting extra comparisons to the callback.

In any case, we can start building this sub-matcher. We can require that the increment step be a unary increment like this:

```
hasIncrement(unaryOperator(hasOperatorName("++")))
```

Specifying what is incremented introduces another quirk of Clang's AST: Usages of variables are represented as `DeclRefExpr`'s ("declaration reference expressions") because they are expressions which refer to variable declarations. To find a `unaryOperator` that refers to a specific declaration, we can simply add a second condition to it:

```
hasIncrement(unaryOperator(
  hasOperatorName("++"),
  hasUnaryOperand(declRefExpr())))
```

Furthermore, we can restrict our matcher to only match if the incremented variable is an integer:

```
hasIncrement(unaryOperator(
  hasOperatorName("++"),
  hasUnaryOperand(declRefExpr(to(varDecl(hasType(isInteger())))))))
```

And the last step will be to attach an identifier to this variable, so that we can retrieve it in the callback:

```
hasIncrement(unaryOperator(
  hasOperatorName("++"),
  hasUnaryOperand(declRefExpr(to(
    varDecl(hasType(isInteger())).bind("incrementVariable"))))))
```

We can add this code to the definition of `LoopMatcher` and make sure that our program, outfitted with the new matcher, only prints out loops that declare a single variable initialized to zero and have an increment step consisting of a unary increment of some variable.

Now, we just need to add a matcher to check if the condition part of the `for` loop compares a variable against the size of the array. There is only one problem - we don't know which array we're iterating over without looking at the body of the loop! We are again restricted to approximating the result we want with matchers, filling in the details in the callback. So we start with:

```
hasCondition(binaryOperator(hasOperatorName("<"))
```

It makes sense to ensure that the left-hand side is a reference to a variable, and that the right-hand side has integer type.

```
hasCondition(binaryOperator(
  hasOperatorName("<"),
  hasLHS(declRefExpr(to(varDecl(hasType(isInteger()))))),
  hasRHS(expr(hasType(isInteger())))))
```

Why? Because it doesn't work. Of the three loops provided in `test-files/simple.cpp`, zero of them have a matching condition. A quick look at the AST dump of the first for loop, produced by the previous iteration of loop-convert, shows us the answer:

```
(ForStmt 0x173b240
  (DeclStmt 0x173afc8
    0x173af50 "int i =
      (IntegerLiteral 0x173afa8 'int' 0)")
  <<<>>>
  (BinaryOperator 0x173b060 '_Bool' '<'
    (ImplicitCastExpr 0x173b030 'int'
      (DeclRefExpr 0x173afe0 'int' lvalue Var 0x173af50 'i' 'int'))
    (ImplicitCastExpr 0x173b048 'int'
      (DeclRefExpr 0x173b008 'const int' lvalue Var 0x170fa80 'N' 'const int')))
  (UnaryOperator 0x173b0b0 'int' lvalue prefix '++'
    (DeclRefExpr 0x173b088 'int' lvalue Var 0x173af50 'i' 'int'))
  (CompoundStatement ...
```

We already know that the declaration and increments both match, or this loop wouldn't have been dumped. The culprit lies in the implicit cast applied to the first operand (i.e. the LHS) of the less-than operator, an L-value to R-value conversion applied to the expression referencing `i`. Thankfully, the matcher library offers a solution to this

problem in the form of `ignoringParenImpCasts`, which instructs the matcher to ignore implicit casts and parentheses before continuing to match. Adjusting the condition operator will restore the desired match.

```
hasCondition(binaryOperator(
  hasOperatorName("<"),
  hasLHS(ignoringParenImpCasts(declRefExpr(
    to(varDecl(hasType(isInteger()))))),
  hasRHS(expr(hasType(isInteger())))))
```

After adding binds to the expressions we wished to capture and extracting the identifier strings into variables, we have array-step-2 completed.

Step 4: Retrieving Matched Nodes

So far, the matcher callback isn't very interesting: it just dumps the loop's AST. At some point, we will need to make changes to the input source code. Next, we'll work on using the nodes we bound in the previous step.

The `MatchFinder::run()` callback takes a `MatchFinder::MatchResult&` as its parameter. We're most interested in its `Context` and `Nodes` members. Clang uses the `ASTContext` class to represent contextual information about the AST, as the name implies, though the most functionally important detail is that several operations require an `ASTContext*` parameter. More immediately useful is the set of matched nodes, and how we retrieve them.

Since we bind three variables (identified by ConditionVarName, InitVarName, and IncrementVarName), we can obtain the matched nodes by using the `getNodeAs()` member function.

In `LoopConvert.cpp` add

```
#include "clang/AST/ASTContext.h"
```

Change `LoopMatcher` to

```
StatementMatcher LoopMatcher =
    forStmt(hasLoopInit(declStmt(
                hasSingleDecl(varDecl(hasInitializer(integerLiteral(equals(0))))
                              .bind("initVarName")))),
            hasIncrement(unaryOperator(
                hasOperatorName("++"),
                hasUnaryOperand(declRefExpr(
                    to(varDecl(hasType(isInteger())).bind("incVarName")))))),
            hasCondition(binaryOperator(
                hasOperatorName("<"),
                hasLHS(ignoringParenImpCasts(declRefExpr(
                    to(varDecl(hasType(isInteger())).bind("condVarName"))))),
                hasRHS(expr(hasType(isInteger()))).bind("forLoop");
```

And change `LoopPrinter::run` to

```
void LoopPrinter::run(const MatchFinder::MatchResult &Result) {
  ASTContext *Context = Result.Context;
  const ForStmt *FS = Result.Nodes.getNodeAs<ForStmt>("forLoop");
  // We do not want to convert header files!
  if (!FS || !Context->getSourceManager().isWrittenInMainFile(FS->getForLoc()))
    return;
  const VarDecl *IncVar = Result.Nodes.getNodeAs<VarDecl>("incVarName");
  const VarDecl *CondVar = Result.Nodes.getNodeAs<VarDecl>("condVarName");
  const VarDecl *InitVar = Result.Nodes.getNodeAs<VarDecl>("initVarName");

  if (!areSameVariable(IncVar, CondVar) || !areSameVariable(IncVar, InitVar))
    return;
  llvm::outs() << "Potential array-based loop discovered.\n";
}
```

Clang associates a `VarDecl` with each variable to represent the variable's declaration. Since the "canonical" form of each declaration is unique by address, all we need to do is make sure neither `ValueDecl` (base class of `VarDecl`) is `NULL` and compare the canonical Decls.

```
static bool areSameVariable(const ValueDecl *First, const ValueDecl *Second) {
  return First && Second &&
         First->getCanonicalDecl() == Second->getCanonicalDecl();
}
```

If execution reaches the end of `LoopPrinter::run()`, we know that the loop shell that looks like

```
for (int i= 0; i < expr(); ++i) { ... }
```

For now, we will just print a message explaining that we found a loop. The next section will deal with recursively traversing the AST to discover all changes needed.

As a side note, it's not as trivial to test if two expressions are the same, though Clang has already done the hard work for us by providing a way to canonicalize expressions:

```
static bool areSameExpr(ASTContext *Context, const Expr *First,
                        const Expr *Second) {
  if (!First || !Second)
    return false;
  llvm::FoldingSetNodeID FirstID, SecondID;
  First->Profile(FirstID, *Context, true);
  Second->Profile(SecondID, *Context, true);
  return FirstID == SecondID;
}
```

This code relies on the comparison between two `llvm::FoldingSetNodeIDs`. As the documentation for `Stmt::Profile()` indicates, the `Profile()` member function builds a description of a node in the AST, based on its properties, along with those of its children. `FoldingSetNodeID` then serves as a hash we can use to compare expressions. We will need `areSameExpr` later. Before you run the new code on the additional loops added to test-files/simple.cpp, try to figure out which ones will be considered potentially convertible.

Matching the Clang AST

This document explains how to use Clang's LibASTMatchers to match interesting nodes of the AST and execute code that uses the matched nodes. Combined with LibTooling, LibASTMatchers helps to write code-to-code transformation tools or query tools.

We assume basic knowledge about the Clang AST. See the Introduction to the Clang AST if you want to learn more about how the AST is structured.

Introduction

LibASTMatchers provides a domain specific language to create predicates on Clang's AST. This DSL is written in and can be used from C++, allowing users to write a single program to both match AST nodes and access the node's C++ interface to extract attributes, source locations, or any other information provided on the AST level.

AST matchers are predicates on nodes in the AST. Matchers are created by calling creator functions that allow building up a tree of matchers, where inner matchers are used to make the match more specific.

For example, to create a matcher that matches all class or union declarations in the AST of a translation unit, you can call recordDecl(). To narrow the match down, for example to find all class or union declarations with the name "`Foo`", insert a hasName matcher: the call `recordDecl(hasName("Foo"))` returns a matcher that matches classes or unions that are named "`Foo`", in any namespace. By default, matchers that accept multiple inner matchers use an implicit allOf(). This allows further narrowing down the match, for example to match all classes that are derived from "`Bar`": `recordDecl(hasName("Foo"), isDerivedFrom("Bar"))`.

How to create a matcher

With more than a thousand classes in the Clang AST, one can quickly get lost when trying to figure out how to create a matcher for a specific pattern. This section will teach you how to use a rigorous step-by-step pattern to build the matcher you are interested in. Note that there will always be matchers missing for some part of the AST. See the section about how to write your own AST matchers later in this document.

The precondition to using the matchers is to understand how the AST for what you want to match looks like. The Introduction to the Clang AST teaches you how to dump a translation unit's AST into a human readable format.

In general, the strategy to create the right matchers is:

1. Find the outermost class in Clang's AST you want to match.
2. Look at the AST Matcher Reference for matchers that either match the node you're interested in or narrow down attributes on the node.
3. Create your outer match expression. Verify that it works as expected.
4. Examine the matchers for what the next inner node you want to match is.
5. Repeat until the matcher is finished.

Binding nodes in match expressions

Matcher expressions allow you to specify which parts of the AST are interesting for a certain task. Often you will want to then do something with the nodes that were matched, like building source code transformations.

To that end, matchers that match specific AST nodes (so called node matchers) are bindable; for example, `recordDecl(hasName("MyClass")).bind("id")` will bind the matched `recordDecl` node to the string "`id`", to be later retrieved in the match callback.

Writing your own matchers

There are multiple different ways to define a matcher, depending on its type and flexibility.

`VariadicDynCastAllOfMatcher<Base, Derived>`

Those match all nodes of type *Base* if they can be dynamically casted to *Derived*. The names of those matchers are nouns, which closely resemble *Derived*. `VariadicDynCastAllOfMatchers` are the backbone of the matcher hierarchy. Most often, your match expression will start with one of them, and you can bind the node they represent to ids for later processing.

`VariadicDynCastAllOfMatchers` are callable classes that model variadic template functions in C++03. They take an arbitrary number of `Matcher<Derived>` and return a `Matcher<Base>`.

`AST_MATCHER_P(Type, Name, ParamType, Param)`

Most matcher definitions use the matcher creation macros. Those define both the matcher of type `Matcher<Type>` itself, and a matcher-creation function named *Name* that takes a parameter of type *ParamType* and returns the corresponding matcher.

There are multiple matcher definition macros that deal with polymorphic return values and different parameter counts. See ASTMatchersMacros.h.

Matcher creation functions

Matchers are generated by nesting calls to matcher creation functions. Most of the time those functions are either created by using `VariadicDynCastAllOfMatcher` or the matcher creation macros (see below). The free-standing functions are an indication that this matcher is just a combination of other matchers, as is for example the case with callee.

Clang Transformer Tutorial

A tutorial on how to write a source-to-source translation tool using Clang Transformer.

What is Clang Transformer? 778
Who is Clang Transformer for? 778
Getting Started 778
Example: style-checking names 779
Example: renaming a function 779
Example: method to function 779
Example: rewriting method calls 780
Reference: ranges, stencils, edits, rules 780
Rewriting ASTs to… Text? 780
Range Selectors 780
Stencils 781
Edits 781
`EditGenerator`s (Advanced) 781
Rules 782
Using a RewriteRule as a clang-tidy check 782
Related Reading 782

What is Clang Transformer?

Clang Transformer is a framework for writing C++ diagnostics and program transformations. It is built on the clang toolchain and the LibTooling library, but aims to hide much of the complexity of clang's native, low-level libraries.

The core abstraction of Transformer is the *rewrite rule*, which specifies how to change a given program pattern into a new form. Here are some examples of tasks you can achieve with Transformer:

- warn against using the name `MkX` for a declared function,
- change `MkX` to `MakeX`, where `MkX` is the name of a declared function,
- change `s.size()` to `Size(s)`, where `s` is a `string`,
- collapse `e.child().m()` to `e.m()`, for any expression `e` and method named `m`.

All of the examples have a common form: they identify a pattern that is the target of the transformation, they specify an *edit* to the code identified by the pattern, and their pattern and edit refer to common variables, like `s`, `e`, and `m`, that range over code fragments. Our first and second examples also specify constraints on the pattern that aren't apparent from the syntax alone, like "`s` is a `string`." Even the first example ("warn …") shares this form, even though it doesn't change any of the code – it's "edit" is simply a no-op.

Transformer helps users succinctly specify rules of this sort and easily execute them locally over a collection of files, apply them to selected portions of a codebase, or even bundle them as a clang-tidy check for ongoing application.

Who is Clang Transformer for?

Clang Transformer is for developers who want to write clang-tidy checks or write tools to modify a large number of C++ files in (roughly) the same way. What qualifies as "large" really depends on the nature of the change and your patience for repetitive editing. In our experience, automated solutions become worthwhile somewhere between 100 and 500 files.

Getting Started

Patterns in Transformer are expressed with clang's AST matchers. Matchers are a language of combinators for describing portions of a clang Abstract Syntax Tree (AST). Since clang's AST includes complete type information (within the limits of single Translation Unit (TU), these patterns can even encode rich constraints on the type properties of AST nodes.

We assume a familiarity with the clang AST and the corresponding AST matchers for the purpose of this tutorial. Users who are unfamiliar with either are encouraged to start with the recommended references in Related Reading.

Example: style-checking names

Assume you have a style-guide rule which forbids functions from being named “MkX” and you want to write a check that catches any violations of this rule. We can express this a Transformer rewrite rule:

```
makeRule(functionDecl(hasName("MkX").bind("fun"),
         noopEdit(node("fun")),
         cat("The name ``MkX`` is not allowed for functions; please rename"));
```

`makeRule` is our go-to function for generating rewrite rules. It takes three arguments: the pattern, the edit, and (optionally) an explanatory note. In our example, the pattern (`functionDecl(...)`) identifies the declaration of the function `MkX`. Since we’re just diagnosing the problem, but not suggesting a fix, our edit is an no-op. But, it contains an *anchor* for the diagnostic message: `node("fun")` says to associate the message with the source range of the AST node bound to “fun”; in this case, the ill-named function declaration. Finally, we use `cat` to build a message that explains the change. Regarding the name `cat` – we’ll discuss it in more detail below, but suffice it to say that it can also take multiple arguments and concatenate their results.

Note that the result of `makeRule` is a value of type `clang::transformer::RewriteRule`, but most users don’t need to care about the details of this type.

Example: renaming a function

Now, let’s extend this example to a *transformation*; specifically, the second example above:

```
makeRule(declRefExpr(to(functionDecl(hasName("MkX")))),
         changeTo(cat("MakeX")),
         cat("MkX has been renamed MakeX"));
```

In this example, the pattern (`declRefExpr(...)`) identifies any *reference* to the function `MkX`, rather than the declaration itself, as in our previous example. Our edit (`changeTo(...)`) says to *change* the code matched by the pattern *to* the text “MakeX”. Finally, we use `cat` again to build a message that explains the change.

Here are some example changes that this rule would make:

Original	Result
`X x = MkX(3);`	`X x = MakeX(3);`
`CallFactory(MkX, 3);`	`CallFactory(MakeX, 3);`
`auto f = MkX;`	`auto f = MakeX;`

Example: method to function

Next, let’s write a rule to replace a method call with a (free) function call, applied to the original method call’s target object. Specifically, “change `s.size()` to `Size(s)`, where `s` is a `string`.” We start with a simpler change that ignores the type of `s`. That is, it will modify *any* method call where the method is named “size”:

```
llvm::StringRef s = "str";
makeRule(
  cxxMemberCallExpr(
    on(expr().bind(s)),
    callee(cxxMethodDecl(hasName("size")))),
  changeTo(cat("Size(", node(s), ")")),
  cat("Method ``size`` is deprecated in favor of free function ``Size``"));
```

We express the pattern with the given AST matcher, which binds the method call’s target to `s` [9]. For the edit, we again use `changeTo`, but this time we construct the term from multiple parts, which we compose with `cat`. The second part of our term is `node(s)`, which selects the source code corresponding to the AST node `s` that was bound when a match was found in the AST for our rule’s pattern. `node(s)` constructs a `RangeSelector`, which, when used in `cat`, indicates that the selected source should be inserted in the output at that point.

Now, we probably don’t want to rewrite *all* invocations of “size” methods, just those on `std::strings`. We can achieve this change simply by refining our matcher. The rest of the rule remains unchanged:

```
llvm::StringRef s = "str";
makeRule(
```

```
  cxxMemberCallExpr(
    on(expr(hasType(namedDecl(hasName("std::string"))))
      .bind(s)),
    callee(cxxMethodDecl(hasName("size")))),
  changeTo(cat("Size(", node(s), ")")),
  cat("Method ``size`` is deprecated in favor of free function ``Size``"));
```

Example: rewriting method calls

In this example, we delete an “intermediary” method call in a string of invocations. This scenario can arise, for example, if you want to collapse a substructure into its parent.

```
llvm::StringRef e = "expr", m = "member";
auto child_call = cxxMemberCallExpr(on(expr().bind(e)),
                                    callee(cxxMethodDecl(hasName("child"))));
makeRule(cxxMemberCallExpr(on(child_call), callee(memberExpr().bind(m)),
         changeTo(cat(e, ".", member(m), "()")),
         cat("``child`` accessor is being removed; call ",
             member(m), " directly on parent"));
```

This rule isn’t quite what we want: it will rewrite `my_object.child().foo()` to `my_object.foo()`, but it will also rewrite `my_ptr->child().foo()` to `my_ptr.foo()`, which is not what we intend. We could fix this by restricting the pattern with `not(isArrow())` in the definition of `child_call`. Yet, we *want* to rewrite calls through pointers.

To capture this idiom, we provide the `access` combinator to intelligently construct a field/method access. In our example, the member access is expressed as:

```
access(e, cat(member(m)))
```

The first argument specifies the object being accessed and the second, a description of the field/method name. In this case, we specify that the method name should be copied from the source – specifically, the source range of `m`’s member. To construct the method call, we would use this expression in `cat`:

```
cat(access(e, cat(member(m))), "()")
```

Reference: ranges, stencils, edits, rules

The above examples demonstrate just the basics of rewrite rules. Every element we touched on has more available constructors: range selectors, stencils, edits and rules. In this section, we’ll briefly review each in turn, with references to the source headers for up-to-date information. First, though, we clarify what rewrite rules are actually rewriting.

Rewriting ASTs to… Text?

The astute reader may have noticed that we’ve been somewhat vague in our explanation of what the rewrite rules are actually rewriting. We’ve referred to “code”, but code can be represented both as raw source text and as an abstract syntax tree. So, which one is it?

Ideally, we’d be rewriting the input AST to a new AST, but clang’s AST is not terribly amenable to this kind of transformation. So, we compromise: we express our patterns and the names that they bind in terms of the AST, but our changes in terms of source code text. We’ve designed Transformer’s language to bridge the gap between the two representations, in an attempt to minimize the user’s need to reason about source code locations and other, low-level syntactic details.

Range Selectors

Transformer provides a small API for describing source ranges: the `RangeSelector` combinators. These ranges are most commonly used to specify the source code affected by an edit and to extract source code in constructing new text.

Roughly, there are two kinds of range combinators: ones that select a source range based on the AST, and others that combine existing ranges into new ranges. For example, `node` selects the range of source spanned by a

particular AST node, as we've seen, while `after` selects the (empty) range located immediately after its argument range. So, `after(node("id"))` is the empty range immediately following the AST node bound to `id`.

For the full collection of `RangeSelectors`, see the header, clang/Tooling/Transformer/RangeSelector.h

Stencils

Transformer offers a large and growing collection of combinators for constructing output. Above, we demonstrated `cat`, the core function for constructing stencils. It takes a series of arguments, of three possible kinds:

1. Raw text, to be copied directly to the output.
2. Selector: specified with a `RangeSelector`, indicates a range of source text to copy to the output.
3. Builder: an operation that constructs a code snippet from its arguments. For example, the `access` function we saw above.

Data of these different types are all represented (generically) by a `Stencil`. `cat` takes text and `RangeSelectors` directly as arguments, rather than requiring that they be constructed with a builder; other builders are constructed explicitly.

In general, `Stencils` produce text from a match result. So, they are not limited to generating source code, but can also be used to generate diagnostic messages that reference (named) elements of the matched code, like we saw in the example of rewriting method calls.

Further details of the `Stencil` type are documented in the header file clang/Tooling/Transformer/Stencil.h.

Edits

Transformer supports additional forms of edits. First, in a `changeTo`, we can specify the particular portion of code to be replaced, using the same `RangeSelector` we saw earlier. For example, we could change the function name in a function declaration with:

```
makeRule(functionDecl(hasName("bad")).bind(f),
         changeTo(name(f), cat("good")),
         cat("bad is now good"));
```

We also provide simpler editing primitives for insertion and deletion: `insertBefore`, `insertAfter` and `remove`. These can all be found in the header file clang/Tooling/Transformer/RewriteRule.h.

We are not limited one edit per match found. Some situations require making multiple edits for each match. For example, suppose we wanted to swap two arguments of a function call.

For this, we provide an overload of `makeRule` that takes a list of edits, rather than just a single one. Our example might look like:

```
makeRule(callExpr(...),
        {changeTo(node(arg0), cat(node(arg2))),
         changeTo(node(arg2), cat(node(arg0)))},
        cat("swap the first and third arguments of the call"));
```

`EditGenerators` (Advanced)

The particular edits we've seen so far are all instances of the `ASTEdit` class, or a list of such. But, not all edits can be expressed as `ASTEdits`. So, we also support a very general signature for edit generators:

```
using EditGenerator = MatchConsumer<llvm::SmallVector<Edit, 1>>;
```

That is, an `EditGenerator` is function that maps a `MatchResult` to a set of edits, or fails. This signature supports a very general form of computation over match results. Transformer provides a number of functions for working with `EditGenerators`, most notably flatten `EditGenerators`, like list flattening. For the full list, see the header file clang/Tooling/Transformer/RewriteRule.h.

Rules

We can also compose multiple *rules*, rather than just edits within a rule, using `applyFirst`: it composes a list of rules as an ordered choice, where Transformer applies the first rule whose pattern matches, ignoring others in the list that follow. If the matchers are independent then order doesn't matter. In that case, `applyFirst` is simply joining the set of rules into one.

The benefit of `applyFirst` is that, for some problems, it allows the user to more concisely formulate later rules in the list, since their patterns need not explicitly exclude the earlier patterns of the list. For example, consider a set of rules that rewrite compound statements, where one rule handles the case of an empty compound statement and the other handles non-empty compound statements. With `applyFirst`, these rules can be expressed compactly as:

```
applyFirst({
  makeRule(compoundStmt(statementCountIs(0)).bind("empty"), ...),
  makeRule(compoundStmt().bind("non-empty"),...)
})
```

The second rule does not need to explicitly specify that the compound statement is non-empty – it follows from the rules position in `applyFirst`. For more complicated examples, this can lead to substantially more readable code.

Sometimes, a modification to the code might require the inclusion of a particular header file. To this end, users can modify rules to specify include directives with `addInclude`.

For additional documentation on these functions, see the header file clang/Tooling/Transformer/RewriteRule.h.

Using a RewriteRule as a clang-tidy check

Transformer supports executing a rewrite rule as a clang-tidy check, with the class `clang::tidy::utils::TransformerClangTidyCheck`. It is designed to require minimal code in the definition. For example, given a rule `MyCheckAsRewriteRule`, one can define a tidy check as follows:

```
class MyCheck : public TransformerClangTidyCheck {
 public:
  MyCheck(StringRef Name, ClangTidyContext *Context)
      : TransformerClangTidyCheck(MyCheckAsRewriteRule, Name, Context) {}
};
```

`TransformerClangTidyCheck` implements the virtual `registerMatchers` and `check` methods based on your rule specification, so you don't need to implement them yourself. If the rule needs to be configured based on the language options and/or the clang-tidy configuration, it can be expressed as a function taking these as parameters and (optionally) returning a `RewriteRule`. This would be useful, for example, for our method-renaming rule, which is parameterized by the original name and the target. For details, see clang-tools-extra/clang-tidy/utils/TransformerClangTidyCheck.h

Related Reading

A good place to start understanding the clang AST and its matchers is with the introductions on clang's site:

- Introduction to the Clang AST
- Matching the Clang AST
- AST Matcher Reference

9 Technically, it binds it to the string "str", to which our variable `s` is bound. But, the choice of that id string is irrelevant, so elide the difference.

ASTImporter: Merging Clang ASTs

The `ASTImporter` class is part of Clang's core library, the AST library. It imports nodes of an `ASTContext` into another `ASTContext`.

In this document, we assume basic knowledge about the Clang AST. See the Introduction to the Clang AST if you want to learn more about how the AST is structured. Knowledge about matching the Clang AST and the reference for the matchers are also useful.

Introduction 783
Algorithm of the import 783
API 784
Errors during the import process 787
Error propagation 788
Polluted AST 788
Using the `-ast-merge` Clang front-end action 790
Example for C 790
Example for C++ 791

Introduction

`ASTContext` holds long-lived AST nodes (such as types and decls) that can be referred to throughout the semantic analysis of a file. In some cases it is preferable to work with more than one `ASTContext`. For example, we'd like to parse multiple different files inside the same Clang tool. It may be convenient if we could view the set of the resulting ASTs as if they were one AST resulting from the parsing of each file together. `ASTImporter` provides the way to copy types or declarations from one `ASTContext` to another. We refer to the context from which we import as the **"from" context** or *source context*; and the context into which we import as the **"to" context** or *destination context*.

Existing clients of the `ASTImporter` library are Cross Translation Unit (CTU) static analysis and the LLDB expression parser. CTU static analysis imports a definition of a function if its definition is found in another translation unit (TU). This way the analysis can breach out from the single TU limitation. LLDB's `expr` command parses a user-defined expression, creates an `ASTContext` for that and then imports the missing definitions from the AST what we got from the debug information (DWARF, etc).

Algorithm of the import

Importing one AST node copies that node into the destination `ASTContext`. Why do we have to copy the node? Isn't enough to insert the pointer to that node into the destination context? One reason is that the "from" context may outlive the "to" context. Also, the Clang AST consider nodes (or certain properties of nodes) equivalent if they have the same address!

The import algorithm has to ensure that the structurally equivalent nodes in the different translation units are not getting duplicated in the merged AST. E.g. if we include the definition of the vector template (`#include <vector>`) in two translation units, then their merged AST should have only one node which represents the template. Also, we have to discover *one definition rule* (ODR) violations. For instance, if there is a class definition with the same name in both translation units, but one of the definition contains a different number of fields. So, we look up existing definitions, and then we check the structural equivalency on those nodes. The following pseudo-code demonstrates the basics of the import mechanism:

```
// Pseudo-code(!) of import:
ErrorOrDecl Import(Decl *FromD) {
  Decl *ToDecl = nullptr;
  FoundDeclsList = Look up all Decls in the "to" Ctx with the same name of FromD;
  for (auto FoundDecl : FoundDeclsList) {
    if (StructurallyEquivalentDecls(FoundDecl, FromD)) {
      ToDecl = FoundDecl;
      Mark FromD as imported;
      break;
    } else {
      Report ODR violation;
      return error;
    }
  }
  if (FoundDeclsList is empty) {
    Import dependent declarations and types of ToDecl;
    ToDecl = create a new AST node in "to" Ctx;
    Mark FromD as imported;
```

```
  }
  return ToDecl;
}
```

Two AST nodes are *structurally equivalent* if they are

- builtin types and refer to the same type, e.g. `int` and `int` are structurally equivalent,
- function types and all their parameters have structurally equivalent types,
- record types and all their fields in order of their definition have the same identifier names and structurally equivalent types,
- variable or function declarations and they have the same identifier name and their types are structurally equivalent.

We could extend the definition of structural equivalency to templates similarly.

If A and B are AST nodes and *A depends on B*, then we say that A is a **dependant** of B and B is a **dependency** of A. The words “dependant” and “dependency” are nouns in British English. Unfortunately, in American English, the adjective “dependent” is used for both meanings. In this document, with the “dependent” adjective we always address the dependencies, the B node in the example.

API

Let’s create a tool which uses the ASTImporter class! First, we build two ASTs from virtual files; the content of the virtual files are synthesized from string literals:

```
std::unique_ptr<ASTUnit> ToUnit = buildASTFromCode(
    "", "to.cc"); // empty file
std::unique_ptr<ASTUnit> FromUnit = buildASTFromCode(
    R"(
    class MyClass {
      int m1;
      int m2;
    };
    )",
    "from.cc");
```

The first AST corresponds to the destination (“to”) context - which is empty - and the second for the source (“from”) context. Next, we define a matcher to match `MyClass` in the “from” context:

```
auto Matcher = cxxRecordDecl(hasName("MyClass"));
auto *From = getFirstDecl<CXXRecordDecl>(Matcher, FromUnit);
```

Now we create the Importer and do the import:

```
ASTImporter Importer(ToUnit->getASTContext(), ToUnit->getFileManager(),
                     FromUnit->getASTContext(), FromUnit->getFileManager(),
                     /*MinimalImport=*/true);
llvm::Expected<Decl *> ImportedOrErr = Importer.Import(From);
```

The `Import` call returns with `llvm::Expected`, so, we must check for any error. Please refer to the error handling documentation for details.

```
if (!ImportedOrErr) {
  llvm::Error Err = ImportedOrErr.takeError();
  llvm::errs() << "ERROR: " << Err << "\n";
  consumeError(std::move(Err));
  return 1;
}
```

If there’s no error then we can get the underlying value. In this example we will print the AST of the “to” context.

```
Decl *Imported = *ImportedOrErr;
Imported->getTranslationUnitDecl()->dump();
```

Since we set **minimal import** in the constructor of the importer, the AST will not contain the declaration of the members (once we run the test tool).

```
TranslationUnitDecl 0x68b9a8 <<invalid sloc>> <invalid sloc>
`-CXXRecordDecl 0x6c7e30 <line:2:7, col:13> col:13 class MyClass definition
  `-DefinitionData pass_in_registers standard_layout trivially_copyable trivial literal
    |-DefaultConstructor exists trivial needs_implicit
    |-CopyConstructor simple trivial has_const_param needs_implicit implicit_has_const_param
    |-MoveConstructor exists simple trivial needs_implicit
    |-CopyAssignment trivial has_const_param needs_implicit implicit_has_const_param
    |-MoveAssignment exists simple trivial needs_implicit
    `-Destructor simple irrelevant trivial needs_implicit
```

We'd like to get the members too, so, we use `ImportDefinition` to copy the whole definition of `MyClass` into the "to" context. Then we dump the AST again.

```
if (llvm::Error Err = Importer.ImportDefinition(From)) {
  llvm::errs() << "ERROR: " << Err << "\n";
  consumeError(std::move(Err));
  return 1;
}
llvm::errs() << "Imported definition.\n";
Imported->getTranslationUnitDecl()->dump();
```

This time the AST is going to contain the members too.

```
TranslationUnitDecl 0x68b9a8 <<invalid sloc>> <invalid sloc>
`-CXXRecordDecl 0x6c7e30 <line:2:7, col:13> col:13 class MyClass definition
  |-DefinitionData pass_in_registers standard_layout trivially_copyable trivial literal
  | |-DefaultConstructor exists trivial needs_implicit
  | |-CopyConstructor simple trivial has_const_param needs_implicit implicit_has_const_param
  | |-MoveConstructor exists simple trivial needs_implicit
  | |-CopyAssignment trivial has_const_param needs_implicit implicit_has_const_param
  | |-MoveAssignment exists simple trivial needs_implicit
  | `-Destructor simple irrelevant trivial needs_implicit
  |-CXXRecordDecl 0x6c7f48 <col:7, col:13> col:13 implicit class MyClass
  |-FieldDecl 0x6c7ff0 <line:3:9, col:13> col:13 m1 'int'
  `-FieldDecl 0x6c8058 <line:4:9, col:13> col:13 m2 'int'
```

We can spare the call for `ImportDefinition` if we set up the importer to do a "normal" (not minimal) import.

```
ASTImporter Importer( ....  /*MinimalImport=*/false);
```

With **normal import**, all dependent declarations are imported normally. However, with minimal import, the dependent Decls are imported without definition, and we have to import their definition for each if we later need that.

Putting this all together here is how the source of the tool looks like:

```
#include "clang/AST/ASTImporter.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"
#include "clang/ASTMatchers/ASTMatchers.h"
#include "clang/Tooling/Tooling.h"

using namespace clang;
using namespace tooling;
using namespace ast_matchers;

template <typename Node, typename Matcher>
Node *getFirstDecl(Matcher M, const std::unique_ptr<ASTUnit> &Unit) {
  auto MB = M.bind("bindStr"); // Bind the to-be-matched node to a string key.
  auto MatchRes = match(MB, Unit->getASTContext());
  // We should have at least one match.
  assert(MatchRes.size() >= 1);
  // Get the first matched and bound node.
  Node *Result =
      const_cast<Node *>(MatchRes[0].template getNodeAs<Node>("bindStr"));
```

```
  assert(Result);
  return Result;
}

int main() {
  std::unique_ptr<ASTUnit> ToUnit = buildASTFromCode(
      "", "to.cc");
  std::unique_ptr<ASTUnit> FromUnit = buildASTFromCode(
      R"(
      class MyClass {
        int m1;
        int m2;
      };
      )",
      "from.cc");
  auto Matcher = cxxRecordDecl(hasName("MyClass"));
  auto *From = getFirstDecl<CXXRecordDecl>(Matcher, FromUnit);

  ASTImporter Importer(ToUnit->getASTContext(), ToUnit->getFileManager(),
                       FromUnit->getASTContext(), FromUnit->getFileManager(),
                       /*MinimalImport=*/true);
  llvm::Expected<Decl *> ImportedOrErr = Importer.Import(From);
  if (!ImportedOrErr) {
    llvm::Error Err = ImportedOrErr.takeError();
    llvm::errs() << "ERROR: " << Err << "\n";
    consumeError(std::move(Err));
    return 1;
  }
  Decl *Imported = *ImportedOrErr;
  Imported->getTranslationUnitDecl()->dump();

  if (llvm::Error Err = Importer.ImportDefinition(From)) {
    llvm::errs() << "ERROR: " << Err << "\n";
    consumeError(std::move(Err));
    return 1;
  }
  llvm::errs() << "Imported definition.\n";
  Imported->getTranslationUnitDecl()->dump();

  return 0;
};
```

We may extend the `CMakeLists.txt` under let's say `clang/tools` with the build and link instructions:

```
add_clang_executable(astimporter-demo ASTImporterDemo.cpp)
clang_target_link_libraries(astimporter-demo
  PRIVATE
  LLVMSupport
  clangAST
  clangASTMatchers
  clangBasic
  clangFrontend
  clangSerialization
  clangTooling
  )
```

Then we can build and execute the new tool.

```
$ ninja astimporter-demo && ./bin/astimporter-demo
```

Errors during the import process

Normally, either the source or the destination context contains the definition of a declaration. However, there may be cases when both of the contexts have a definition for a given symbol. If these definitions differ, then we have a name conflict, in C++ it is known as ODR (one definition rule) violation. Let's modify the previous tool we had written and try to import a `ClassTemplateSpecializationDecl` with a conflicting definition:

```
int main() {
  std::unique_ptr<ASTUnit> ToUnit = buildASTFromCode(
      R"(
      // primary template
      template <typename T>
      struct X {};
      // explicit specialization
      template<>
      struct X<int> { int i; };
      )",
      "to.cc");
  ToUnit->enableSourceFileDiagnostics();
  std::unique_ptr<ASTUnit> FromUnit = buildASTFromCode(
      R"(
      // primary template
      template <typename T>
      struct X {};
      // explicit specialization
      template<>
      struct X<int> { int i2; };
      // field mismatch:  ^^
      )",
      "from.cc");
  FromUnit->enableSourceFileDiagnostics();
  auto Matcher = classTemplateSpecializationDecl(hasName("X"));
  auto *From = getFirstDecl<ClassTemplateSpecializationDecl>(Matcher, FromUnit);
  auto *To = getFirstDecl<ClassTemplateSpecializationDecl>(Matcher, ToUnit);

  ASTImporter Importer(ToUnit->getASTContext(), ToUnit->getFileManager(),
                       FromUnit->getASTContext(), FromUnit->getFileManager(),
                       /*MinimalImport=*/false);
  llvm::Expected<Decl *> ImportedOrErr = Importer.Import(From);
  if (!ImportedOrErr) {
    llvm::Error Err = ImportedOrErr.takeError();
    llvm::errs() << "ERROR: " << Err << "\n";
    consumeError(std::move(Err));
    To->getTranslationUnitDecl()->dump();
    return 1;
  }
  return 0;
};
```

When we run the tool we have the following warning:

```
to.cc:7:14: warning: type 'X<int>' has incompatible definitions in different translation units [-Wodr]
      struct X<int> { int i; };
             ^
to.cc:7:27: note: field has name 'i' here
      struct X<int> { int i; };
                      ^
from.cc:7:27: note: field has name 'i2' here
      struct X<int> { int i2; };
                      ^
```

Note, because of these diagnostics we had to call `enableSourceFileDiagnostics` on the `ASTUnit` objects.

Since we could not import the specified declaration (`From`), we get an error in the return value. The AST does not contain the conflicting definition, so we are left with the original AST.

```
ERROR: NameConflict
TranslationUnitDecl 0xe54a48 <<invalid sloc>> <invalid sloc>
|-ClassTemplateDecl 0xe91020 <to.cc:3:7, line:4:17> col:14 X
| |-TemplateTypeParmDecl 0xe90ed0 <line:3:17, col:26> col:26 typename depth 0 index 0 T
| |-CXXRecordDecl 0xe90f90 <line:4:7, col:17> col:14 struct X definition
| | |-DefinitionData empty aggregate standard_layout trivially_copyable pod trivial literal has_constexpr_non_copy_move_ctor can_const_default_init
| | | |-DefaultConstructor exists trivial constexpr needs_implicit defaulted_is_constexpr
| | | |-CopyConstructor simple trivial has_const_param needs_implicit implicit_has_const_param
| | | |-MoveConstructor exists simple trivial needs_implicit
| | | |-CopyAssignment trivial has_const_param needs_implicit implicit_has_const_param
| | | |-MoveAssignment exists simple trivial needs_implicit
| | | `-Destructor simple irrelevant trivial needs_implicit
| | `-CXXRecordDecl 0xe91270 <col:7, col:14> col:14 implicit struct X
| `-ClassTemplateSpecialization 0xe91340 'X'
`-ClassTemplateSpecializationDecl 0xe91340 <line:6:7, line:7:30> col:14 struct X definition
  |-DefinitionData pass_in_registers aggregate standard_layout trivially_copyable pod trivial literal
  | |-DefaultConstructor exists trivial needs_implicit
  | |-CopyConstructor simple trivial has_const_param needs_implicit implicit_has_const_param
  | |-MoveConstructor exists simple trivial needs_implicit
  | |-CopyAssignment trivial has_const_param needs_implicit implicit_has_const_param
  | |-MoveAssignment exists simple trivial needs_implicit
  | `-Destructor simple irrelevant trivial needs_implicit
  |-TemplateArgument type 'int'
  |-CXXRecordDecl 0xe91558 <col:7, col:14> col:14 implicit struct X
  `-FieldDecl 0xe91600 <col:23, col:27> col:27 i 'int'
```

Error propagation

If there is a dependent node we have to import before we could import a given node then the import error associated to the dependency propagates to the dependant node. Let's modify the previous example and import a `FieldDecl` instead of the `ClassTemplateSpecializationDecl`.

```
auto Matcher = fieldDecl(hasName("i2"));
auto *From = getFirstDecl<FieldDecl>(Matcher, FromUnit);
```

In this case we can see that an error is associated (`getImportDeclErrorIfAny`) to the specialization also, not just to the field:

```
llvm::Expected<Decl *> ImportedOrErr = Importer.Import(From);
if (!ImportedOrErr) {
  llvm::Error Err = ImportedOrErr.takeError();
  consumeError(std::move(Err));

  // check that the ClassTemplateSpecializationDecl is also marked as
  // erroneous.
  auto *FromSpec = getFirstDecl<ClassTemplateSpecializationDecl>(
      classTemplateSpecializationDecl(hasName("X")), FromUnit);
  assert(Importer.getImportDeclErrorIfAny(FromSpec));
  // Btw, the error is also set for the FieldDecl.
  assert(Importer.getImportDeclErrorIfAny(From));
  return 1;
}
```

Polluted AST

We may recognize an error during the import of a dependent node. However, by that time, we had already created the dependant. In these cases we do not remove the existing erroneous node from the "to" context, rather we associate an error to that node. Let's extend the previous example with another class `Y`. This class has a forward definition in the "to" context, but its definition is in the "from" context. We'd like to import the definition, but it contains a member whose type conflicts with the type in the "to" context:

```
std::unique_ptr<ASTUnit> ToUnit = buildASTFromCode(
    R"(
    // primary template
    template <typename T>
    struct X {};
    // explicit specialization
    template<>
    struct X<int> { int i; };
```

```
    class Y;
    )",
    "to.cc");
ToUnit->enableSourceFileDiagnostics();
std::unique_ptr<ASTUnit> FromUnit = buildASTFromCode(
    R"(
    // primary template
    template <typename T>
    struct X {};
    // explicit specialization
    template<>
    struct X<int> { int i2; };
    // field mismatch:  ^^

    class Y { void f() { X<int> xi; } };
    )",
    "from.cc");
FromUnit->enableSourceFileDiagnostics();
auto Matcher = cxxRecordDecl(hasName("Y"));
auto *From = getFirstDecl<CXXRecordDecl>(Matcher, FromUnit);
auto *To = getFirstDecl<CXXRecordDecl>(Matcher, ToUnit);
```

This time we create a shared_ptr for `ASTImporterSharedState` which owns the associated errors for the “to” context. Note, there may be several different ASTImporter objects which import into the same “to” context but from different “from” contexts; they should share the same `ASTImporterSharedState`. (Also note, we have to include the corresponding `ASTImporterSharedState.h` header file.)

```
auto ImporterState = std::make_shared<ASTImporterSharedState>();
ASTImporter Importer(ToUnit->getASTContext(), ToUnit->getFileManager(),
                     FromUnit->getASTContext(), FromUnit->getFileManager(),
                     /*MinimalImport=*/false, ImporterState);
llvm::Expected<Decl *> ImportedOrErr = Importer.Import(From);
if (!ImportedOrErr) {
  llvm::Error Err = ImportedOrErr.takeError();
  consumeError(std::move(Err));

  // ... but the node had been created.
  auto *ToYDef = getFirstDecl<CXXRecordDecl>(
      cxxRecordDecl(hasName("Y"), isDefinition()), ToUnit);
  ToYDef->dump();
  // An error is set for "ToYDef" in the shared state.
  Optional<ImportError> OptErr =
      ImporterState->getImportDeclErrorIfAny(ToYDef);
  assert(OptErr);

  return 1;
}
```

If we take a look at the AST, then we can see that the Decl with the definition is created, but the field is missing.

```
|-CXXRecordDecl 0xf66678 <line:9:7, col:13> col:13 class Y
`-CXXRecordDecl 0xf66730 prev 0xf66678 <:10:7, col:13> col:13 class Y definition
  |-DefinitionData pass_in_registers empty aggregate standard_layout trivially_copyable pod trivial literal has_constexpr_non_copy_move_ctor can_const_default_init
  | |-DefaultConstructor exists trivial constexpr needs_implicit defaulted_is_constexpr
  | |-CopyConstructor simple trivial has_const_param needs_implicit implicit_has_const_param
  | |-MoveConstructor exists simple trivial needs_implicit
  | |-CopyAssignment trivial has_const_param needs_implicit implicit_has_const_param
  | |-MoveAssignment exists simple trivial needs_implicit
  | `-Destructor simple irrelevant trivial needs_implicit
  `-CXXRecordDecl 0xf66828 <col:7, col:13> col:13 implicit class Y
```

We do not remove the erroneous nodes because by the time when we recognize the error it is too late to remove the node, there may be additional references to that already in the AST. This is aligned with the overall design principle of the Clang AST: Clang AST nodes (types, declarations, statements, expressions, and so on) are generally designed to be **immutable once created**. Thus, clients of the ASTImporter library should always check if there is any associated error for the node which they inspect in the destination context. We recommend skipping the processing of those nodes which have an error associated with them.

Using the `-ast-merge` Clang front-end action

The `-ast-merge <pch-file>` command-line switch can be used to merge from the given serialized AST file. This file represents the source context. When this switch is present then each top-level AST node of the source context is being merged into the destination context. If the merge was successful then `ASTConsumer::HandleTopLevelDecl` is called for the Decl. This results that we can execute the original front-end action on the extended AST.

Example for C

Let's consider the following three files:

```
// bar.h
#ifndef BAR_H
#define BAR_H
int bar();
#endif /* BAR_H */

// bar.c
#include "bar.h"
int bar() {
  return 41;
}

// main.c
#include "bar.h"
int main() {
    return bar();
}
```

Let's generate the AST files for the two source files:

```
$ clang -cc1 -emit-pch -o bar.ast bar.c
$ clang -cc1 -emit-pch -o main.ast main.c
```

Then, let's check how the merged AST would look like if we consider only the `bar()` function:

```
$ clang -cc1 -ast-merge bar.ast -ast-merge main.ast /dev/null -ast-dump
TranslationUnitDecl 0x12b0738 <<invalid sloc>> <invalid sloc>
|-FunctionDecl 0x12b1470 </path/bar.h:4:1, col:9> col:5 used bar 'int ()'
|-FunctionDecl 0x12b1538 prev 0x12b1470 </path/bar.c:3:1, line:5:1> line:3:5 used bar 'int ()'
| `-CompoundStmt 0x12b1608 <col:11, line:5:1>
|   `-ReturnStmt 0x12b15f8 <line:4:3, col:10>
|     `-IntegerLiteral 0x12b15d8 <col:10> 'int' 41
|-FunctionDecl 0x12b1648 prev 0x12b1538 </path/bar.h:4:1, col:9> col:5 used bar 'int ()'
```

We can inspect that the prototype of the function and the definition of it is merged into the same redeclaration chain. What's more there is a third prototype declaration merged to the chain. The functions are merged in a way that prototypes are added to the redecl chain if they refer to the same type, but we can have only one definition. The first two declarations are from `bar.ast`, the third is from `main.ast`.

Now, let's create an object file from the merged AST:

```
$ clang -cc1 -ast-merge bar.ast -ast-merge main.ast /dev/null -emit-obj -o main.o
```

Next, we may call the linker and execute the created binary file.

```
$ clang -o a.out main.o
$ ./a.out
$ echo $?
41
$
```

Example for C++

In the case of C++, the generation of the AST files and the way how we invoke the front-end is a bit different. Assuming we have these three files:

```
// foo.h
#ifndef FOO_H
#define FOO_H
struct foo {
    virtual int fun();
};
#endif /* FOO_H */

// foo.cpp
#include "foo.h"
int foo::fun() {
  return 42;
}

// main.cpp
#include "foo.h"
int main() {
    return foo().fun();
}
```

We shall generate the AST files, merge them, create the executable and then run it:

```
$ clang++ -x c++-header -o foo.ast foo.cpp
$ clang++ -x c++-header -o main.ast main.cpp
$ clang++ -cc1 -x c++ -ast-merge foo.ast -ast-merge main.ast /dev/null -ast-dump
$ clang++ -cc1 -x c++ -ast-merge foo.ast -ast-merge main.ast /dev/null -emit-obj -o main.o
$ clang++ -o a.out main.o
$ ./a.out
$ echo $?
42
$
```

How To Setup Clang Tooling For LLVM

Clang Tooling provides infrastructure to write tools that need syntactic and semantic information about a program. This term also relates to a set of specific tools using this infrastructure (e.g. `clang-check`). This document provides information on how to set up and use Clang Tooling for the LLVM source code.

Introduction

Clang Tooling needs a compilation database to figure out specific build options for each file. Currently it can create a compilation database from the `compile_commands.json` file, generated by CMake. When invoking clang tools, you can either specify a path to a build directory using a command line parameter `-p` or let Clang Tooling find this file in your source tree. In either case you need to configure your build using CMake to use clang tools.

Setup Clang Tooling Using CMake and Make

If you intend to use make to build LLVM, you should have CMake 2.8.6 or later installed (can be found here).

First, you need to generate Makefiles for LLVM with CMake. You need to make a build directory and run CMake from it:

```
$ mkdir your/build/directory
$ cd your/build/directory
$ cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON path/to/llvm/sources
```

If you want to use clang instead of GCC, you can add `-DCMAKE_C_COMPILER=/path/to/clang -DCMAKE_CXX_COMPILER=/path/to/clang++`. You can also use `ccmake`, which provides a curses interface to configure CMake variables.

As a result, the new `compile_commands.json` file should appear in the current directory. You should link it to the LLVM source tree so that Clang Tooling is able to use it:

```
$ ln -s $PWD/compile_commands.json path/to/llvm/source/
```

Now you are ready to build and test LLVM using make:

```
$ make check-all
```

Setup Clang Tooling Using CMake on Windows

For Windows developers, the Visual Studio project generators in CMake do not support CMAKE_EXPORT_COMPILE_COMMANDS. However, the Ninja generator does support this variable and can be used on Windows to generate a suitable `compile_commands.json` that invokes the MSVC compiler.

First, you will need to install Ninja. Once installed, the Ninja executable will need to be in your search path for CMake to locate it.

Next, assuming you already have Visual Studio installed on your machine, you need to have the appropriate environment variables configured so that CMake will locate the MSVC compiler for the Ninja generator. The documentation describes the necessary environment variable settings, but the simplest thing is to use a developer command-prompt window or call a developer command file to set the environment variables appropriately.

Now you can run CMake with the Ninja generator to export a compilation database:

```
C:\> mkdir build-ninja
C:\> cd build-ninja
C:\build-ninja> cmake -G Ninja -DCMAKE_EXPORT_COMPILE_COMMANDS=ON path/to/llvm/sources
```

It is best to keep your Visual Studio IDE build folder separate from the Ninja build folder. This prevents the two build systems from negatively interacting with each other.

Once the `compile_commands.json` file has been created by Ninja, you can use that compilation database with Clang Tooling. One caveat is that because there are indirect settings obtained through the environment variables, you may need to run any Clang Tooling executables through a command prompt window created for use with Visual Studio as described above. An alternative, e.g. for using the Visual Studio debugger on a Clang Tooling executable, is to ensure that the environment variables are also visible to the debugger settings. This can be done locally in Visual Studio's debugger configuration locally or globally by launching the Visual Studio IDE from a suitable command-prompt window.

Using Clang Tools

After you completed the previous steps, you are ready to run clang tools. If you have a recent clang installed, you should have `clang-check` in `$PATH`. Try to run it on any `.cpp` file inside the LLVM source tree:

```
$ clang-check tools/clang/lib/Tooling/CompilationDatabase.cpp
```

If you're using vim, it's convenient to have clang-check integrated. Put this into your `.vimrc`:

```
function! ClangCheckImpl(cmd)
  if &autowrite | wall | endif
  echo "Running " . a:cmd . " ..."
  let l:output = system(a:cmd)
  cexpr l:output
  cwindow
  let w:quickfix_title = a:cmd
  if v:shell_error != 0
    cc
  endif
  let g:clang_check_last_cmd = a:cmd
endfunction
```

```
function! ClangCheck()
  let l:filename = expand('%')
  if l:filename =~ '\.\(cpp\|cxx\|cc\|c\)$'
    call ClangCheckImpl("clang-check " . l:filename)
  elseif exists("g:clang_check_last_cmd")
    call ClangCheckImpl(g:clang_check_last_cmd)
  else
    echo "Can't detect file's compilation arguments and no previous clang-check invocation!"
  endif
endfunction

nmap <silent> <F5> :call ClangCheck()<CR><CR>
```

When editing a .cpp/.cxx/.cc/.c file, hit F5 to reparse the file. In case the current file has a different extension (for example, .h), F5 will re-run the last clang-check invocation made from this vim instance (if any). The output will go into the error window, which is opened automatically when clang-check finds errors, and can be re-opened with `:cope`.

Other `clang-check` options that can be useful when working with clang AST:

- `-ast-print` — Build ASTs and then pretty-print them.
- `-ast-dump` — Build ASTs and then debug dump them.
- `-ast-dump-filter=<string>` — Use with `-ast-dump` or `-ast-print` to dump/print only AST declaration nodes having a certain substring in a qualified name. Use `-ast-list` to list all filterable declaration node names.
- `-ast-list` — Build ASTs and print the list of declaration node qualified names.

Examples:

```
$ clang-check tools/clang/tools/clang-check/ClangCheck.cpp -ast-dump -ast-dump-filter ActionFactory::newASTConsumer
Processing: tools/clang/tools/clang-check/ClangCheck.cpp.
Dumping ::ActionFactory::newASTConsumer:
clang::ASTConsumer *newASTConsumer() (CompoundStmt 0x44da290 </home/alexfh/local/llvm/tools/clang/tools/clang-check/ClangCheck.cpp:64:40, line:72:3>
  (IfStmt 0x44d97c8 <line:65:5, line:66:45>
    <<<NULL>>>
      (ImplicitCastExpr 0x44d96d0 <line:65:9> '_Bool':'_Bool' <UserDefinedConversion>
...
$ clang-check tools/clang/tools/clang-check/ClangCheck.cpp -ast-print -ast-dump-filter ActionFactory::newASTConsumer
Processing: tools/clang/tools/clang-check/ClangCheck.cpp.
Printing <anonymous namespace>::ActionFactory::newASTConsumer:
clang::ASTConsumer *newASTConsumer() {
    if (this->ASTList.operator _Bool())
        return clang::CreateASTDeclNodeLister();
    if (this->ASTDump.operator _Bool())
        return clang::CreateASTDumper(nullptr /*Dump to stdout.*/,
                                      this->ASTDumpFilter);
    if (this->ASTPrint.operator _Bool())
        return clang::CreateASTPrinter(&llvm::outs(), this->ASTDumpFilter);
    return new clang::ASTConsumer();
}
```

Using Ninja Build System

Optionally you can use the Ninja build system instead of make. It is aimed at making your builds faster. Currently this step will require building Ninja from sources.

To take advantage of using Clang Tools along with Ninja build you need at least CMake 2.8.9.

Clone the Ninja git repository and build Ninja from sources:

```
$ git clone git://github.com/martine/ninja.git
$ cd ninja/
$ ./bootstrap.py
```

This will result in a single binary `ninja` in the current directory. It doesn't require installation and can just be copied to any location inside `$PATH`, say `/usr/local/bin/`:

```
$ sudo cp ninja /usr/local/bin/
$ sudo chmod a+rx /usr/local/bin/ninja
```

After doing all of this, you'll need to generate Ninja build files for LLVM with CMake. You need to make a build directory and run CMake from it:

```
$ mkdir your/build/directory
$ cd your/build/directory
$ cmake -G Ninja -DCMAKE_EXPORT_COMPILE_COMMANDS=ON path/to/llvm/sources
```

If you want to use clang instead of GCC, you can add `-DCMAKE_C_COMPILER=/path/to/clang -DCMAKE_CXX_COMPILER=/path/to/clang++`. You can also use `ccmake`, which provides a curses interface to configure CMake variables in an interactive manner.

As a result, the new `compile_commands.json` file should appear in the current directory. You should link it to the LLVM source tree so that Clang Tooling is able to use it:

```
$ ln -s $PWD/compile_commands.json path/to/llvm/source/
```

Now you are ready to build and test LLVM using Ninja:

```
$ ninja check-all
```

Other target names can be used in the same way as with make.

JSON Compilation Database Format Specification

This document describes a format for specifying how to replay single compilations independently of the build system.

Background

Tools based on the C++ Abstract Syntax Tree need full information how to parse a translation unit. Usually this information is implicitly available in the build system, but running tools as part of the build system is not necessarily the best solution:

- Build systems are inherently change driven, so running multiple tools over the same code base without changing the code does not fit into the architecture of many build systems.
- Figuring out whether things have changed is often an IO bound process; this makes it hard to build low latency end user tools based on the build system.
- Build systems are inherently sequential in the build graph, for example due to generated source code. While tools that run independently of the build still need the generated source code to exist, running tools multiple times over unchanging source does not require serialization of the runs according to the build dependency graph.

Supported Systems

Clang has the ablity to generate compilation database fragments via the **`-MJ argument`**. You can concatenate those fragments together between `[` and `]` to create a compilation database.

Currently CMake (since 2.8.5) supports generation of compilation databases for Unix Makefile builds (Ninja builds in the works) with the option `CMAKE_EXPORT_COMPILE_COMMANDS`.

For projects on Linux, there is an alternative to intercept compiler calls with a tool called Bear.

Bazel can export a compilation database via this extractor extension. Bazel is otherwise resistant to Bear and other compiler-intercept techniques.

Clang's tooling interface supports reading compilation databases; see the LibTooling documentation. libclang and its python bindings also support this (since clang 3.2); see CXCompilationDatabase.h.

Format

A compilation database is a JSON file, which consist of an array of "command objects", where each command object specifies one way a translation unit is compiled in the project.

Each command object contains the translation unit's main file, the working directory of the compile run and the actual compile command.

Example:

```
[
  { "directory": "/home/user/llvm/build",
    "arguments": ["/usr/bin/clang++", "-Irelative", "-DSOMEDEF=With spaces, quotes and \\-es.", "-c", "-o", "file.o", "file.cc"],
    "file": "file.cc" },

  { "directory": "/home/user/llvm/build",
    "command": "/usr/bin/clang++ -Irelative -DSOMEDEF=\"With spaces, quotes and \\-es.\" -c -o file.o file.cc",
    "file": "file2.cc" },

  ...
]
```

The contracts for each field in the command object are:

- **directory:** The working directory of the compilation. All paths specified in the **command** or **file** fields must be either absolute or relative to this directory.
- **file:** The main translation unit source processed by this compilation step. This is used by tools as the key into the compilation database. There can be multiple command objects for the same file, for example if the same source file is compiled with different configurations.
- **arguments:** The compile command argv as list of strings. This should run the compilation step for the translation unit `file`. `arguments[0]` should be the executable name, such as `clang++`. Arguments should not be escaped, but ready to pass to `execvp()`.
- **command:** The compile command as a single shell-escaped string. Arguments may be shell quoted and escaped following platform conventions, with ‘"’ and ‘\’ being the only special characters. Shell expansion is not supported.

  Either **arguments** or **command** is required. **arguments** is preferred, as shell (un)escaping is a possible source of errors.
- **output:** The name of the output created by this compilation step. This field is optional. It can be used to distinguish different processing modes of the same input file.

Build System Integration

The convention is to name the file compile_commands.json and put it at the top of the build directory. Clang tools are pointed to the top of the build directory to detect the file and use the compilation database to parse C++ code in the source tree.

Alternatives

For simple projects, Clang tools also recognize a `compile_flags.txt` file. This should contain one argument per line. The same flags will be used to compile any file.

Example:

```
-xc++
-I
libwidget/include/
```

Here `-I libwidget/include` is two arguments, and so becomes two lines. Paths are relative to the directory containing `compile_flags.txt`.

Clang’s refactoring engine

This document describes the design of Clang’s refactoring engine and provides a couple of examples that show how various primitives in the refactoring API can be used to implement different refactoring actions. The LibTooling library provides several other APIs that are used when developing a refactoring action.

Refactoring engine can be used to implement local refactorings that are initiated using a selection in an editor or an IDE. You can combine AST matchers and the refactoring engine to implement refactorings that don’t lend themselves well to source selection and/or have to query ASTs for some particular nodes.

We assume basic knowledge about the Clang AST. See the Introduction to the Clang AST if you want to learn more about how the AST is structured.

Introduction

Clang’s refactoring engine defines a set refactoring actions that implement a number of different source transformations. The `clang-refactor` command-line tool can be used to perform these refactorings. Certain refactorings are also available in other clients like text editors and IDEs.

A refactoring action is a class that defines a list of related refactoring operations (rules). These rules are grouped under a common umbrella - a single `clang-refactor` command. In addition to rules, the refactoring action provides the action’s command name and description to `clang-refactor`. Each action must implement the `RefactoringAction` interface. Here’s an outline of a `local-rename` action:

```
class LocalRename final : public RefactoringAction {
public:
  StringRef getCommand() const override { return "local-rename"; }

  StringRef getDescription() const override {
    return "Finds and renames symbols in code with no indexer support";
  }

  RefactoringActionRules createActionRules() const override {
    ...
  }
};
```

Refactoring Action Rules

An individual refactoring action is responsible for creating the set of grouped refactoring action rules that represent one refactoring operation. Although the rules in one action may have a number of different implementations, they should strive to produce a similar result. It should be easy for users to identify which refactoring action produced the result regardless of which refactoring action rule was used.

The distinction between actions and rules enables the creation of actions that define a set of different rules that produce similar results. For example, the “add missing switch cases” refactoring operation typically adds missing cases to one switch at a time. However, it could be useful to have a refactoring that works on all switches that operate on a particular enum, as one could then automatically update all of them after adding a new enum constant. To achieve that, we can create two different rules that will use one `clang-refactor` subcommand. The first rule will describe a local operation that’s initiated when the user selects a single switch. The second rule will describe a global operation that works across translation units and is initiated when the user provides the name of the enum to clang-refactor (or the user could select the enum declaration instead). The clang-refactor tool will then analyze the selection and other options passed to the refactoring action, and will pick the most appropriate rule for the given selection and other options.

Rule Types

Clang’s refactoring engine supports several different refactoring rules:

- `SourceChangeRefactoringRule` produces source replacements that are applied to the source files. Subclasses that choose to implement this rule have to implement the `createSourceReplacements` member function. This type of rule is typically used to implement local refactorings that transform the source in one translation unit only.
- `FindSymbolOccurrencesRefactoringRule` produces a “partial” refactoring result: a set of occurrences that refer to a particular symbol. This type of rule is typically used to implement an interactive renaming action that allows users to specify which occurrences should be renamed during the refactoring. Subclasses that choose to implement this rule have to implement the `findSymbolOccurrences` member function.

The following set of quick checks might help if you are unsure about the type of rule you should use:

1. If you would like to transform the source in one translation unit and if you don’t need any cross-TU information, then the `SourceChangeRefactoringRule` should work for you.
2. If you would like to implement a rename-like operation with potential interactive components, then `FindSymbolOccurrencesRefactoringRule` might work for you.

How to Create a Rule

Once you determine which type of rule is suitable for your needs you can implement the refactoring by subclassing the rule and implementing its interface. The subclass should have a constructor that takes the inputs that are needed to perform the refactoring. For example, if you want to implement a rule that simply deletes a selection, you should create a subclass of `SourceChangeRefactoringRule` with a constructor that accepts the selection range:

```
class DeleteSelectedRange final : public SourceChangeRefactoringRule {
public:
  DeleteSelection(SourceRange Selection) : Selection(Selection) {}

  Expected<AtomicChanges>
  createSourceReplacements(RefactoringRuleContext &Context) override {
    AtomicChange Replacement(Context.getSources(), Selection.getBegin());
    Replacement.replace(Context.getSource,
                        CharSourceRange::getCharRange(Selection), "");
    return { Replacement };
  }
private:
  SourceRange Selection;
};
```

The rule's subclass can then be added to the list of refactoring action's rules for a particular action using the `createRefactoringActionRule` function. For example, the class that's shown above can be added to the list of action rules using the following code:

```
RefactoringActionRules Rules;
Rules.push_back(
  createRefactoringActionRule<DeleteSelectedRange>(
        SourceRangeSelectionRequirement())
);
```

The `createRefactoringActionRule` function takes in a list of refactoring action rule requirement values. These values describe the initiation requirements that have to be satisfied by the refactoring engine before the provided action rule can be constructed and invoked. The next section describes how these requirements are evaluated and lists all the possible requirements that can be used to construct a refactoring action rule.

Refactoring Action Rule Requirements

A refactoring action rule requirement is a value whose type derives from the `RefactoringActionRuleRequirement` class. The type must define an `evaluate` member function that returns a value of type `Expected<...>`. When a requirement value is used as an argument to `createRefactoringActionRule`, that value is evaluated during the initiation of the action rule. The evaluated result is then passed to the rule's constructor unless the evaluation produced an error. For example, the `DeleteSelectedRange` sample rule that's defined in the previous section will be evaluated using the following steps:

1. `SourceRangeSelectionRequirement`'s `evaluate` member function will be called first. It will return an `Expected<SourceRange>`.
2. If the return value is an error the initiation will fail and the error will be reported to the client. Note that the client may not report the error to the user.
3. Otherwise the source range return value will be used to construct the `DeleteSelectedRange` rule. The rule will then be invoked as the initiation succeeded (all requirements were evaluated successfully).

The same series of steps applies to any refactoring rule. Firstly, the engine will evaluate all of the requirements. Then it will check if these requirements are satisfied (they should not produce an error). Then it will construct the rule and invoke it.

The separation of requirements, their evaluation and the invocation of the refactoring action rule allows the refactoring clients to:

- Disable refactoring action rules whose requirements are not supported.

- Gather the set of options and define a command-line / visual interface that allows users to input these options without ever invoking the action.

Selection Requirements

The refactoring rule requirements that require some form of source selection are listed below:

- `SourceRangeSelectionRequirement` evaluates to a source range when the action is invoked with some sort of selection. This requirement should be satisfied when a refactoring is initiated in an editor, even when the user has not selected anything (the range will contain the cursor's location in that case).

Other Requirements

There are several other requirements types that can be used when creating a refactoring rule:

- The `RefactoringOptionsRequirement` requirement is an abstract class that should be subclassed by requirements working with options. The more concrete `OptionRequirement` requirement is a simple implementation of the aforementioned class that returns the value of the specified option when it's evaluated. The next section talks more about refactoring options and how they can be used when creating a rule.

Refactoring Options

Refactoring options are values that affect a refactoring operation and are specified either using command-line options or another client-specific mechanism. Options should be created using a class that derives either from the `OptionalRequiredOption` or `RequiredRefactoringOption`. The following example shows how one can created a required string option that corresponds to the `-new-name` command-line option in clang-refactor:

```
class NewNameOption : public RequiredRefactoringOption<std::string> {
public:
  StringRef getName() const override { return "new-name"; }
  StringRef getDescription() const override {
    return "The new name to change the symbol to";
  }
};
```

The option that's shown in the example above can then be used to create a requirement for a refactoring rule using a requirement like `OptionRequirement`:

```
createRefactoringActionRule<RenameOccurrences>(
  ...,
  OptionRequirement<NewNameOption>())
);
```

Using Clang Tools

Overview

Clang Tools are standalone command line (and potentially GUI) tools designed for use by C++ developers who are already using and enjoying Clang as their compiler. These tools provide developer-oriented functionality such as fast syntax checking, automatic formatting, refactoring, etc.

Only a couple of the most basic and fundamental tools are kept in the primary Clang tree. The rest of the tools are kept in a separate directory tree, clang-tools-extra.

This document describes a high-level overview of the organization of Clang Tools within the project as well as giving an introduction to some of the more important tools. However, it should be noted that this document is currently focused on Clang and Clang Tool developers, not on end users of these tools.

Clang Tools Organization

Clang Tools are CLI or GUI programs that are intended to be directly used by C++ developers. That is they are *not* primarily for use by Clang developers, although they are hopefully useful to C++ developers who happen to work on Clang, and we try to actively dogfood their functionality. They are developed in three components: the underlying infrastructure for building a standalone tool based on Clang, core shared logic used by many different tools in the form of refactoring and rewriting libraries, and the tools themselves.

The underlying infrastructure for Clang Tools is the LibTooling platform. See its documentation for much more detailed information about how this infrastructure works. The common refactoring and rewriting toolkit-style library is also part of LibTooling organizationally.

A few Clang Tools are developed along side the core Clang libraries as examples and test cases of fundamental functionality. However, most of the tools are developed in a side repository to provide easy separation from the core libraries. We intentionally do not support public libraries in the side repository, as we want to carefully review and find good APIs for libraries as they are lifted out of a few tools and into the core Clang library set.

Regardless of which repository Clang Tools' code resides in, the development process and practices for all Clang Tools are exactly those of Clang itself. They are entirely within the Clang *project*, regardless of the version control scheme.

Core Clang Tools

The core set of Clang tools that are within the main repository are tools that very specifically complement, and allow use and testing of *Clang* specific functionality.

`clang-check`

ClangCheck combines the LibTooling framework for running a Clang tool with the basic Clang diagnostics by syntax checking specific files in a fast, command line interface. It can also accept flags to re-display the diagnostics in different formats with different flags, suitable for use driving an IDE or editor. Furthermore, it can be used in fixit-mode to directly apply fixit-hints offered by clang. See How To Setup Clang Tooling For LLVM for instructions on how to setup and used *clang-check*.

`clang-format`

Clang-format is both a library and a stand-alone tool with the goal of automatically reformatting C++ sources files according to configurable style guides. To do so, clang-format uses Clang's `Lexer` to transform an input file into a token stream and then changes all the whitespace around those tokens. The goal is for clang-format to serve both as a user tool (ideally with powerful IDE integrations) and as part of other refactoring tools, e.g. to do a reformatting of all the lines changed during a renaming.

Extra Clang Tools

As various categories of Clang Tools are added to the extra repository, they'll be tracked here. The focus of this documentation is on the scope and features of the tools for other tool developers; each tool should provide its own user-focused documentation.

`clang-tidy`

clang-tidy is a clang-based C++ linter tool. It provides an extensible framework for building compiler-based static analyses detecting and fixing bug-prone patterns, performance, portability and maintainability issues.

Ideas for new Tools

- C++ cast conversion tool. Will convert C-style casts (`(type) value`) to appropriate C++ cast (`static_cast`, `const_cast` or `reinterpret_cast`).
- Non-member `begin()` and `end()` conversion tool. Will convert `foo.begin()` into `begin(foo)` and similarly for `end()`, where `foo` is a standard container. We could also detect similar patterns for arrays.

- `tr1` removal tool. Will migrate source code from using TR1 library features to C++11 library. For example:

```
#include <tr1/unordered_map>
int main()
{
    std::tr1::unordered_map <int, int> ma;
    std::cout << ma.size () << std::endl;
    return 0;
}
```

  should be rewritten to:

```
#include <unordered_map>
int main()
{
    std::unordered_map <int, int> ma;
    std::cout << ma.size () << std::endl;
    return 0;
}
```

- A tool to remove `auto`. Will convert `auto` to an explicit type or add comments with deduced types. The motivation is that there are developers that don't want to use `auto` because they are afraid that they might lose control over their code.
- C++14: less verbose operator function objects (N3421). For example:

```
sort(v.begin(), v.end(), greater<ValueType>());
```

  should be rewritten to:

```
sort(v.begin(), v.end(), greater<>());
```

ClangCheck

ClangCheck is a small wrapper around LibTooling which can be used to do basic error checking and AST dumping.

```
$ cat <<EOF > snippet.cc
> void f() {
>   int a = 0
> }
> EOF
$ ~/clang/build/bin/clang-check snippet.cc -ast-dump --
Processing: /Users/danieljasper/clang/llvm/tools/clang/docs/snippet.cc.
/Users/danieljasper/clang/llvm/tools/clang/docs/snippet.cc:2:12: error: expected ';' at end of
      declaration
  int a = 0
           ^
           ;
(TranslationUnitDecl 0x7ff3a3029ed0 <<invalid sloc>>
  (TypedefDecl 0x7ff3a302a410 <<invalid sloc>> __int128_t '__int128')
  (TypedefDecl 0x7ff3a302a470 <<invalid sloc>> __uint128_t 'unsigned __int128')
  (TypedefDecl 0x7ff3a302a830 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]')
  (FunctionDecl 0x7ff3a302a8d0 </Users/danieljasper/clang/llvm/tools/clang/docs/snippet.cc:1:1, line:3:1> f 'void (void)'
    (CompoundStmt 0x7ff3a302aa10 <line:1:10, line:3:1>
      (DeclStmt 0x7ff3a302a9f8 <line:2:3, line:3:1>
        (VarDecl 0x7ff3a302a980 <line:2:3, col:11> a 'int'
          (IntegerLiteral 0x7ff3a302a9d8 <col:11> 'int' 0))))))
1 error generated.
Error while processing snippet.cc.
```

The '–' at the end is important as it prevents **clang-check** from searching for a compilation database. For more information on how to setup and use **clang-check** in a project, see How To Setup Clang Tooling For LLVM.

ClangFormat

ClangFormat describes a set of tools that are built on top of LibFormat. It can support your workflow in a variety of ways including a standalone tool and editor integrations.

Standalone Tool

clang-format is located in *clang/tools/clang-format* and can be used to format C/C++/Java/JavaScript/JSON/Objective-C/Protobuf/C# code.

```
$ clang-format -help
OVERVIEW: A tool to format C/C++/Java/JavaScript/JSON/Objective-C/Protobuf/C# code.

If no arguments are specified, it formats the code from standard input
and writes the result to the standard output.
If <file>s are given, it reformats the files. If -i is specified
together with <file>s, the files are edited in-place. Otherwise, the
result is written to the standard output.

USAGE: clang-format [options] [<file> ...]

OPTIONS:

Clang-format options:

  --Werror                        - If set, changes formatting warnings to errors
  --Wno-error=<value>             - If set don't error out on the specified warning type.
    =unknown                      -   If set, unknown format options are only warned about.
                                      This can be used to enable formatting, even if the
                                      configuration contains unknown (newer) options.
                                      Use with caution, as this might lead to dramatically
                                      differing format depending on an option being
                                      supported or not.
  --assume-filename=<string>      - Override filename used to determine the language.
                                    When reading from stdin, clang-format assumes this
                                    filename to determine the language.
  --cursor=<uint>                 - The position of the cursor when invoking
                                    clang-format from an editor integration
  --dry-run                       - If set, do not actually make the formatting changes
  --dump-config                   - Dump configuration options to stdout and exit.
                                    Can be used with -style option.
  --fallback-style=<string>       - The name of the predefined style used as a
                                    fallback in case clang-format is invoked with
                                    -style=file, but can not find the .clang-format
                                    file to use.
                                    Use -fallback-style=none to skip formatting.
  --ferror-limit=<uint>           - Set the maximum number of clang-format errors to emit
                                    before stopping (0 = no limit).
                                    Used only with --dry-run or -n
  --files=<string>                - Provide a list of files to run clang-format
  -i                              - Inplace edit <file>s, if specified.
  --length=<uint>                 - Format a range of this length (in bytes).
                                    Multiple ranges can be formatted by specifying
                                    several -offset and -length pairs.
                                    When only a single -offset is specified without
                                    -length, clang-format will format up to the end
                                    of the file.
                                    Can only be used with one input file.
  --lines=<string>                - <start line>:<end line> - format a range of
                                    lines (both 1-based).
                                    Multiple ranges can be formatted by specifying
                                    several -lines arguments.
                                    Can't be used with -offset and -length.
                                    Can only be used with one input file.
  -n                              - Alias for --dry-run
  --offset=<uint>                 - Format a range starting at this byte offset.
                                    Multiple ranges can be formatted by specifying
                                    several -offset and -length pairs.
                                    Can only be used with one input file.
  --output-replacements-xml       - Output replacements as XML.
  --qualifier-alignment=<string>  - If set, overrides the qualifier alignment style
                                    determined by the QualifierAlignment style flag
```

```
  --sort-includes              - If set, overrides the include sorting behavior
                                 determined by the SortIncludes style flag
  --style=<string>             - Coding style, currently supports:
                                   LLVM, GNU, Google, Chromium, Microsoft, Mozilla, WebKit.
                                 Use -style=file to load style configuration from
                                 .clang-format file located in one of the parent
                                 directories of the source file (or current
                                 directory for stdin).
                                 Use -style=file:<format_file_path> to explicitly specify
                                 the configuration file.
                                 Use -style="{key: value, ...}" to set specific
                                 parameters, e.g.:
                                   -style="{BasedOnStyle: llvm, IndentWidth: 8}"
  --verbose                    - If set, shows the list of processed files

Generic Options:

  --help                       - Display available options (--help-hidden for more)
  --help-list                  - Display list of available options (--help-list-hidden for more)
  --version                    - Display the version of this program
```

When the desired code formatting style is different from the available options, the style can be customized using the `-style="{key: value, ...}"` option or by putting your style configuration in the `.clang-format` or `_clang-format` file in your project's directory and using `clang-format -style=file`.

An easy way to create the `.clang-format` file is:

```
clang-format -style=llvm -dump-config > .clang-format
```

Available style options are described in Clang-Format Style Options.

Vim Integration

There is an integration for **vim** which lets you run the **clang-format** standalone tool on your current buffer, optionally selecting regions to reformat. The integration has the form of a *python*-file which can be found under *clang/tools/clang-format/clang-format.py*.

This can be integrated by adding the following to your *.vimrc*:

```
map <C-K> :pyf <path-to-this-file>/clang-format.py<cr>
imap <C-K> <c-o>:pyf <path-to-this-file>/clang-format.py<cr>
```

The first line enables **clang-format** for NORMAL and VISUAL mode, the second line adds support for INSERT mode. Change "C-K" to another binding if you need **clang-format** on a different key (C-K stands for Ctrl+k).

With this integration you can press the bound key and clang-format will format the current line in NORMAL and INSERT mode or the selected region in VISUAL mode. The line or region is extended to the next bigger syntactic entity.

It operates on the current, potentially unsaved buffer and does not create or save any files. To revert a formatting, just undo.

An alternative option is to format changes when saving a file and thus to have a zero-effort integration into the coding workflow. To do this, add this to your *.vimrc*:

```
function! Formatonsave()
  let l:formatdiff = 1
  pyf ~/llvm/tools/clang/tools/clang-format/clang-format.py
endfunction
autocmd BufWritePre *.h,*.cc,*.cpp call Formatonsave()
```

Emacs Integration

Similar to the integration for **vim**, there is an integration for **emacs**. It can be found at *clang/tools/clang-format/clang-format.el* and used by adding this to your *.emacs*:

```
(load "<path-to-clang>/tools/clang-format/clang-format.el")
(global-set-key [C-M-tab] 'clang-format-region)
```

This binds the function *clang-format-region* to C-M-tab, which then formats the current line or selected region.

BBEdit Integration

clang-format cannot be used as a text filter with BBEdit, but works well via a script. The AppleScript to do this integration can be found at *clang/tools/clang-format/clang-format-bbedit.applescript*; place a copy in *~/Library/Application Support/BBEdit/Scripts*, and edit the path within it to point to your local copy of **clang-format**.

With this integration you can select the script from the Script menu and **clang-format** will format the selection. Note that you can rename the menu item by renaming the script, and can assign the menu item a keyboard shortcut in the BBEdit preferences, under Menus & Shortcuts.

CLion Integration

clang-format is integrated into CLion as an alternative code formatter. CLion turns it on automatically when there is a `.clang-format` file under the project root. Code style rules are applied as you type, including indentation, auto-completion, code generation, and refactorings.

clang-format can also be enabled without a `.clang-format` file. In this case, CLion prompts you to create one based on the current IDE settings or the default LLVM style.

Visual Studio Integration

Download the latest Visual Studio extension from the alpha build site. The default key-binding is Ctrl-R,Ctrl-F.

Visual Studio Code Integration

Get the latest Visual Studio Code extension from the Visual Studio Marketplace. The default key-binding is Alt-Shift-F.

Script for patch reformatting

The python script *clang/tools/clang-format/clang-format-diff.py* parses the output of a unified diff and reformats all contained lines with **clang-format**.

```
usage: clang-format-diff.py [-h] [-i] [-p NUM] [-regex PATTERN] [-style STYLE]

Reformat changed lines in diff. Without -i option just output the diff that
would be introduced.

optional arguments:
  -h, --help      show this help message and exit
  -i              apply edits to files instead of displaying a diff
  -p NUM          strip the smallest prefix containing P slashes
  -regex PATTERN  custom pattern selecting file paths to reformat
  -style STYLE    formatting style to apply (LLVM, Google, Chromium, Mozilla,
                  WebKit)
```

So to reformat all the lines in the latest **git** commit, just do:

```
git diff -U0 --no-color HEAD^ | clang-format-diff.py -i -p1
```

With Mercurial/**hg**:

```
hg diff -U0 --color=never | clang-format-diff.py -i -p1
```

In an SVN client, you can do:

```
svn diff --diff-cmd=diff -x -U0 | clang-format-diff.py -i
```

The option *-U0* will create a diff without context lines (the script would format those as well).

These commands use the file paths shown in the diff output so they will only work from the root of the repository.

Current State of Clang Format for LLVM

The following table Clang Formatted Status shows the current status of clang-formatting for the entire LLVM source tree.

Clang-Format Style Options

Clang-Format Style Options describes configurable formatting style options supported by LibFormat and ClangFormat.

When using **clang-format** command line utility or `clang::format::reformat(...)` functions from code, one can either use one of the predefined styles (LLVM, Google, Chromium, Mozilla, WebKit, Microsoft) or create a custom style by configuring specific style options.

Configuring Style with clang-format

clang-format supports two ways to provide custom style options: directly specify style configuration in the `-style=` command line option or use `-style=file` and put style configuration in the `.clang-format` or `_clang-format` file in the project directory.

When using `-style=file`, **clang-format** for each input file will try to find the `.clang-format` file located in the closest parent directory of the input file. When the standard input is used, the search is started from the current directory.

When using `-style=file:<format_file_path>`, **clang-format** for each input file will use the format file located at *<format_file_path>*. The path may be absolute or relative to the working directory.

The `.clang-format` file uses YAML format:

```
key1: value1
key2: value2
# A comment.
...
```

The configuration file can consist of several sections each having different `Language:` parameter denoting the programming language this section of the configuration is targeted at. See the description of the **Language** option below for the list of supported languages. The first section may have no language set, it will set the default style options for all languages. Configuration sections for specific language will override options set in the default section.

When **clang-format** formats a file, it auto-detects the language using the file name. When formatting standard input or a file that doesn't have the extension corresponding to its language, `-assume-filename=` option can be used to override the file name **clang-format** uses to detect the language.

An example of a configuration file for multiple languages:

```
---
# We'll use defaults from the LLVM style, but with 4 columns indentation.
BasedOnStyle: LLVM
IndentWidth: 4
---
Language: Cpp
# Force pointers to the type for C++.
DerivePointerAlignment: false
PointerAlignment: Left
---
Language: JavaScript
```

```
# Use 100 columns for JS.
ColumnLimit: 100
---
Language: Proto
# Don't format .proto files.
DisableFormat: true
---
Language: CSharp
# Use 100 columns for C#.
ColumnLimit: 100
...
```

An easy way to get a valid `.clang-format` file containing all configuration options of a certain predefined style is:

```
clang-format -style=llvm -dump-config > .clang-format
```

When specifying configuration in the `-style=` option, the same configuration is applied for all input files. The format of the configuration is:

```
-style='{key1: value1, key2: value2, ...}'
```

Disabling Formatting on a Piece of Code

Clang-format understands also special comments that switch formatting in a delimited range. The code between a comment `// clang-format off` or `/* clang-format off */` up to a comment `// clang-format on` or `/* clang-format on */` will not be formatted. The comments themselves will be formatted (aligned) normally.

```
int formatted_code;
// clang-format off
    void    unformatted_code  ;
// clang-format on
void formatted_code_again;
```

Configuring Style in Code

When using `clang::format::reformat(...)` functions, the format is specified by supplying the clang::format::FormatStyle structure.

Configurable Format Style Options

This section lists the supported style options. Value type is specified for each option. For enumeration types possible values are specified both as a C++ enumeration member (with a prefix, e.g. `LS_Auto`), and as a value usable in the configuration (without a prefix: `Auto`).

BasedOnStyle (`String`)

The style used for all options not specifically set in the configuration.

This option is supported only in the **clang-format** configuration (both within `-style='{...}'` and the `.clang-format` file).

Possible values:

- `LLVM` A style complying with the LLVM coding standards
- `Google` A style complying with Google's C++ style guide
- `Chromium` A style complying with Chromium's style guide
- `Mozilla` A style complying with Mozilla's style guide
- `WebKit` A style complying with WebKit's style guide
- `Microsoft` A style complying with Microsoft's style guide
- `GNU` A style complying with the GNU coding standards

- `InheritParentConfig` Not a real style, but allows to use the `.clang-format` file from the parent directory (or its parent if there is none). If there is no parent file found it falls back to the `fallback` style, and applies the changes to that.

  With this option you can overwrite some parts of your main style for your subdirectories. This is also possible through the command line, e.g.: `--style={BasedOnStyle: InheritParentConfig, ColumnLimit: 20}`

AccessModifierOffset (`Integer`) clang-format 3.3

The extra indent or outdent of access modifiers, e.g. `public:`.

AlignAfterOpenBracket (`BracketAlignmentStyle`) clang-format 3.8

If `true`, horizontally aligns arguments after an open bracket.

This applies to round brackets (parentheses), angle brackets and square brackets.

Possible values:

- `BAS_Align` (in configuration: `Align`) Align parameters on the open bracket, e.g.:

```
someLongFunction(argument1,
                 argument2);
```

- `BAS_DontAlign` (in configuration: `DontAlign`) Don't align, instead use `ContinuationIndentWidth`, e.g.:

```
someLongFunction(argument1,
    argument2);
```

- `BAS_AlwaysBreak` (in configuration: `AlwaysBreak`) Always break after an open bracket, if the parameters don't fit on a single line, e.g.:

```
someLongFunction(
    argument1, argument2);
```

- `BAS_BlockIndent` (in configuration: `BlockIndent`) Always break after an open bracket, if the parameters don't fit on a single line. Closing brackets will be placed on a new line. E.g.:

```
someLongFunction(
    argument1, argument2
)
```

> **Warning**
>
> Note: This currently only applies to parentheses.

AlignArrayOfStructures (`ArrayInitializerAlignmentStyle`) clang-format 13

if not `None`, when using initialization for an array of structs aligns the fields into columns.

NOTE: As of clang-format 15 this option only applied to arrays with equal number of columns per row.

Possible values:

- `AIAS_Left` (in configuration: `Left`) Align array column and left justify the columns e.g.:

```
struct test demo[] =
{
    {56, 23,    "hello"},
    {-1, 93463, "world"},
    {7,  5,     "!!"   }
};
```

- `AIAS_Right` (in configuration: `Right`) Align array column and right justify the columns e.g.:

```
struct test demo[] =
{
```

```
        {56,    23, "hello"},
        {-1, 93463, "world"},
        { 7,     5,    "!!"}
    };
```

- `AIAS_None` (in configuration: `None`) Don't align array initializer columns.

AlignConsecutiveAssignments (`AlignConsecutiveStyle`) clang-format 3.8

Style of aligning consecutive assignments.

`Consecutive` will result in formattings like:

```
int a            = 1;
int somelongname = 2;
double c         = 3;
```

Nested configuration flags:

Alignment options.

They can also be read as a whole for compatibility. The choices are: - None - Consecutive - AcrossEmptyLines - AcrossComments - AcrossEmptyLinesAndComments

For example, to align across empty lines and not across comments, either of these work.

```
AlignConsecutiveMacros: AcrossEmptyLines

AlignConsecutiveMacros:
  Enabled: true
  AcrossEmptyLines: true
  AcrossComments: false
```

- `bool Enabled` Whether aligning is enabled.

```
#define SHORT_NAME       42
#define LONGER_NAME      0x007f
#define EVEN_LONGER_NAME (2)
#define foo(x)           (x * x)
#define bar(y, z)        (y + z)

int a            = 1;
int somelongname = 2;
double c         = 3;

int aaaa : 1;
int b    : 12;
int ccc  : 8;

int         aaaa = 12;
float       b = 23;
std::string ccc;
```

- `bool AcrossEmptyLines` Whether to align across empty lines.

```
true:
int a            = 1;
int somelongname = 2;
double c         = 3;

int d            = 3;

false:
int a            = 1;
int somelongname = 2;
```

```
double c         = 3;

int d = 3;
```

- `bool AcrossComments` Whether to align across comments.

```
true:
int d    = 3;
/* A comment. */
double e = 4;

false:
int d = 3;
/* A comment. */
double e = 4;
```

- `bool AlignCompound` Only for `AlignConsecutiveAssignments`. Whether compound assignments like += are aligned along with =.

```
true:
a   &= 2;
bbb  = 2;

false:
a &= 2;
bbb = 2;
```

- `bool PadOperators` Only for `AlignConsecutiveAssignments`. Whether short assignment operators are left-padded to the same length as long ones in order to put all assignment operators to the right of the left hand side.

```
true:
a   >>= 2;
bbb   = 2;

a     = 2;
bbb >>= 2;

false:
a >>= 2;
bbb = 2;

a     = 2;
bbb >>= 2;
```

AlignConsecutiveBitFields (`AlignConsecutiveStyle`) clang-format 11

Style of aligning consecutive bit fields.

`Consecutive` will align the bitfield separators of consecutive lines. This will result in formattings like:

```
int aaaa : 1;
int b    : 12;
int ccc  : 8;
```

Nested configuration flags:

Alignment options.

They can also be read as a whole for compatibility. The choices are: - None - Consecutive - AcrossEmptyLines - AcrossComments - AcrossEmptyLinesAndComments

For example, to align across empty lines and not across comments, either of these work.

```
AlignConsecutiveMacros: AcrossEmptyLines

AlignConsecutiveMacros:
```

```
Enabled: true
AcrossEmptyLines: true
AcrossComments: false
```

- `bool Enabled` Whether aligning is enabled.

```
#define SHORT_NAME       42
#define LONGER_NAME      0x007f
#define EVEN_LONGER_NAME (2)
#define foo(x)           (x * x)
#define bar(y, z)        (y + z)

int a            = 1;
int somelongname = 2;
double c         = 3;

int aaaa : 1;
int b    : 12;
int ccc  : 8;

int         aaaa = 12;
float       b = 23;
std::string ccc;
```

- `bool AcrossEmptyLines` Whether to align across empty lines.

```
true:
int a            = 1;
int somelongname = 2;
double c         = 3;

int d            = 3;

false:
int a            = 1;
int somelongname = 2;
double c         = 3;

int d = 3;
```

- `bool AcrossComments` Whether to align across comments.

```
true:
int d    = 3;
/* A comment. */
double e = 4;

false:
int d = 3;
/* A comment. */
double e = 4;
```

- `bool AlignCompound` Only for `AlignConsecutiveAssignments`. Whether compound assignments like += are aligned along with =.

```
true:
a   &= 2;
bbb  = 2;

false:
a &= 2;
bbb = 2;
```

- `bool PadOperators` Only for `AlignConsecutiveAssignments`. Whether short assignment operators are left-padded to the same length as long ones in order to put all assignment operators to the right of the left hand side.

```
true:
a   >>= 2;
bbb   = 2;

a     = 2;
bbb >>= 2;

false:
a >>= 2;
bbb = 2;

a     = 2;
bbb >>= 2;
```

AlignConsecutiveDeclarations (`AlignConsecutiveStyle`) clang-format 3.8

Style of aligning consecutive declarations.

`Consecutive` will align the declaration names of consecutive lines. This will result in formattings like:

```
int         aaaa = 12;
float       b = 23;
std::string ccc;
```

Nested configuration flags:

Alignment options.

They can also be read as a whole for compatibility. The choices are: - None - Consecutive - AcrossEmptyLines - AcrossComments - AcrossEmptyLinesAndComments

For example, to align across empty lines and not across comments, either of these work.

```
AlignConsecutiveMacros: AcrossEmptyLines

AlignConsecutiveMacros:
  Enabled: true
  AcrossEmptyLines: true
  AcrossComments: false
```

- `bool Enabled` Whether aligning is enabled.

```
#define SHORT_NAME       42
#define LONGER_NAME      0x007f
#define EVEN_LONGER_NAME (2)
#define foo(x)           (x * x)
#define bar(y, z)        (y + z)

int a            = 1;
int somelongname = 2;
double c         = 3;

int aaaa : 1;
int b    : 12;
int ccc  : 8;

int         aaaa = 12;
float       b = 23;
std::string ccc;
```

- `bool AcrossEmptyLines` Whether to align across empty lines.

```
true:
int a            = 1;
int somelongname = 2;
double c         = 3;

int d            = 3;

false:
int a            = 1;
int somelongname = 2;
double c         = 3;

int d = 3;
```

- `bool AcrossComments` Whether to align across comments.

```
true:
int d    = 3;
/* A comment. */
double e = 4;

false:
int d = 3;
/* A comment. */
double e = 4;
```

- `bool AlignCompound` Only for `AlignConsecutiveAssignments`. Whether compound assignments like += are aligned along with =.

```
true:
a   &= 2;
bbb  = 2;

false:
a &= 2;
bbb = 2;
```

- `bool PadOperators` Only for `AlignConsecutiveAssignments`. Whether short assignment operators are left-padded to the same length as long ones in order to put all assignment operators to the right of the left hand side.

```
true:
a   >>= 2;
bbb   = 2;

a     = 2;
bbb >>= 2;

false:
a >>= 2;
bbb = 2;

a     = 2;
bbb >>= 2;
```

AlignConsecutiveMacros (`AlignConsecutiveStyle`) clang-format 9

Style of aligning consecutive macro definitions.

`Consecutive` will result in formattings like:

```
#define SHORT_NAME       42
#define LONGER_NAME      0x007f
#define EVEN_LONGER_NAME (2)
#define foo(x)           (x * x)
```

```
#define bar(y, z)        (y + z)
```

Nested configuration flags:

Alignment options.

They can also be read as a whole for compatibility. The choices are: - None - Consecutive - AcrossEmptyLines - AcrossComments - AcrossEmptyLinesAndComments

For example, to align across empty lines and not across comments, either of these work.

```
AlignConsecutiveMacros: AcrossEmptyLines

AlignConsecutiveMacros:
  Enabled: true
  AcrossEmptyLines: true
  AcrossComments: false
```

- `bool Enabled` Whether aligning is enabled.

  ```
  #define SHORT_NAME       42
  #define LONGER_NAME      0x007f
  #define EVEN_LONGER_NAME (2)
  #define foo(x)           (x * x)
  #define bar(y, z)        (y + z)

  int a            = 1;
  int somelongname = 2;
  double c         = 3;

  int aaaa : 1;
  int b    : 12;
  int ccc  : 8;

  int         aaaa = 12;
  float       b = 23;
  std::string ccc;
  ```

- `bool AcrossEmptyLines` Whether to align across empty lines.

  ```
  true:
  int a            = 1;
  int somelongname = 2;
  double c         = 3;

  int d            = 3;

  false:
  int a            = 1;
  int somelongname = 2;
  double c         = 3;

  int d = 3;
  ```

- `bool AcrossComments` Whether to align across comments.

  ```
  true:
  int d    = 3;
  /* A comment. */
  double e = 4;

  false:
  ```

```
int d = 3;
/* A comment. */
double e = 4;
```

- `bool AlignCompound` Only for `AlignConsecutiveAssignments`. Whether compound assignments like += are aligned along with =.

```
true:
a   &= 2;
bbb  = 2;

false:
a &= 2;
bbb = 2;
```

- `bool PadOperators` Only for `AlignConsecutiveAssignments`. Whether short assignment operators are left-padded to the same length as long ones in order to put all assignment operators to the right of the left hand side.

```
true:
a   >>= 2;
bbb   = 2;

a     = 2;
bbb >>= 2;

false:
a >>= 2;
bbb = 2;

a     = 2;
bbb >>= 2;
```

AlignEscapedNewlines (`EscapedNewlineAlignmentStyle`) clang-format 5

Options for aligning backslashes in escaped newlines.

Possible values:

- `ENAS_DontAlign` (in configuration: `DontAlign`) Don't align escaped newlines.

```
#define A \
  int aaaa; \
  int b; \
  int ddddddddd;
```

- `ENAS_Left` (in configuration: `Left`) Align escaped newlines as far left as possible.

```
true:
#define A   \
  int aaaa; \
  int b;    \
  int ddddddddd;

false:
```

- `ENAS_Right` (in configuration: `Right`) Align escaped newlines in the right-most column.

```
#define A                                                                      \
  int aaaa;                                                                    \
  int b;                                                                       \
  int ddddddddd;
```

AlignOperands (`OperandAlignmentStyle`) clang-format 3.5

If `true`, horizontally align operands of binary and ternary expressions.

Possible values:

- `OAS_DontAlign` (in configuration: `DontAlign`) Do not align operands of binary and ternary expressions. The wrapped lines are indented `ContinuationIndentWidth` spaces from the start of the line.
- `OAS_Align` (in configuration: `Align`) Horizontally align operands of binary and ternary expressions.

  Specifically, this aligns operands of a single expression that needs to be split over multiple lines, e.g.:

```
int aaa = bbbbbbbbbbbbbbb +
          ccccccccccccccc;
```

  When `BreakBeforeBinaryOperators` is set, the wrapped operator is aligned with the operand on the first line.

```
int aaa = bbbbbbbbbbbbbbb
          + ccccccccccccccc;
```

- `OAS_AlignAfterOperator` (in configuration: `AlignAfterOperator`) Horizontally align operands of binary and ternary expressions.

  This is similar to `AO_Align`, except when `BreakBeforeBinaryOperators` is set, the operator is un-indented so that the wrapped operand is aligned with the operand on the first line.

```
int aaa = bbbbbbbbbbbbbbb
        + ccccccccccccccc;
```

AlignTrailingComments (`Boolean`) clang-format 3.7

If `true`, aligns trailing comments.

```
true:                                   false:
int a;     // My comment a      vs.     int a; // My comment a
int b = 2; // comment  b                int b = 2; // comment about b
```

AllowAllArgumentsOnNextLine (`Boolean`) clang-format 9

If a function call or braced initializer list doesn't fit on a line, allow putting all arguments onto the next line, even if `BinPackArguments` is `false`.

```
true:
callFunction(
    a, b, c, d);

false:
callFunction(a,
             b,
             c,
             d);
```

AllowAllConstructorInitializersOnNextLine (`Boolean`) clang-format 9

This option is **deprecated**. See `NextLine` of `PackConstructorInitializers`.

AllowAllParametersOfDeclarationOnNextLine (`Boolean`) clang-format 3.3

If the function declaration doesn't fit on a line, allow putting all parameters of a function declaration onto the next line even if `BinPackParameters` is `false`.

```
true:
void myFunction(
    int a, int b, int c, int d, int e);

false:
void myFunction(int a,
                int b,
                int c,
                int d,
                int e);
```

AllowShortBlocksOnASingleLine (`ShortBlockStyle`) clang-format 3.5

Dependent on the value, `while (true) { continue; }` can be put on a single line.

Possible values:

- `SBS_Never` (in configuration: `Never`) Never merge blocks into a single line.

```
while (true) {
}
while (true) {
  continue;
}
```

- `SBS_Empty` (in configuration: `Empty`) Only merge empty blocks.

```
while (true) {}
while (true) {
  continue;
}
```

- `SBS_Always` (in configuration: `Always`) Always merge short blocks into a single line.

```
while (true) {}
while (true) { continue; }
```

AllowShortCaseLabelsOnASingleLine (`Boolean`) clang-format 3.6

If `true`, short case labels will be contracted to a single line.

```
true:                                   vs.     false:
switch (a) {                                    switch (a) {
case 1: x = 1; break;                           case 1:
case 2: return;                                   x = 1;
}                                                 break;
                                                case 2:
                                                  return;
                                                }
```

AllowShortEnumsOnASingleLine (`Boolean`) clang-format 11

Allow short enums on a single line.

```
true:
enum { A, B } myEnum;

false:
enum {
  A,
  B
} myEnum;
```

AllowShortFunctionsOnASingleLine (`ShortFunctionStyle`) clang-format 3.5

Dependent on the value, `int f() { return 0; }` can be put on a single line.

Possible values:

- `SFS_None` (in configuration: `None`) Never merge functions into a single line.
- `SFS_InlineOnly` (in configuration: `InlineOnly`) Only merge functions defined inside a class. Same as “inline”, except it does not implies “empty”: i.e. top level empty functions are not merged either.

```
class Foo {
  void f() { foo(); }
};
void f() {
  foo();
}
void f() {
}
```

- `SFS_Empty` (in configuration: `Empty`) Only merge empty functions.

```
void f() {}
void f2() {
  bar2();
}
```

- `SFS_Inline` (in configuration: `Inline`) Only merge functions defined inside a class. Implies “empty”.

```
class Foo {
  void f() { foo(); }
};
void f() {
  foo();
}
void f() {}
```

- `SFS_All` (in configuration: `All`) Merge all functions fitting on a single line.

```
class Foo {
  void f() { foo(); }
};
void f() { bar(); }
```

AllowShortIfStatementsOnASingleLine (`ShortIfStyle`**)** clang-format 3.3

Dependent on the value, `if (a) return;` can be put on a single line.

Possible values:

- `SIS_Never` (in configuration: `Never`) Never put short ifs on the same line.

```
if (a)
  return;

if (b)
  return;
else
  return;

if (c)
  return;
else {
  return;
}
```

- `SIS_WithoutElse` (in configuration: `WithoutElse`) Put short ifs on the same line only if there is no else statement.

```
if (a) return;

if (b)
  return;
else
  return;

if (c)
  return;
else {
  return;
}
```

- `SIS_OnlyFirstIf` (in configuration: `OnlyFirstIf`) Put short ifs, but not else ifs nor else statements, on the same line.

```
if (a) return;
```

```
if (b) return;
else if (b)
  return;
else
  return;

if (c) return;
else {
  return;
}
```

- `SIS_AllIfsAndElse` (in configuration: `AllIfsAndElse`) Always put short ifs, else ifs and else statements on the same line.

```
if (a) return;

if (b) return;
else return;

if (c) return;
else {
  return;
}
```

AllowShortLambdasOnASingleLine (`ShortLambdaStyle`**)** clang-format 9

Dependent on the value, `auto lambda []() { return 0; }` can be put on a single line.

Possible values:

- `SLS_None` (in configuration: `None`) Never merge lambdas into a single line.
- `SLS_Empty` (in configuration: `Empty`) Only merge empty lambdas.

```
auto lambda = [](int a) {}
auto lambda2 = [](int a) {
    return a;
};
```

- `SLS_Inline` (in configuration: `Inline`) Merge lambda into a single line if argument of a function.

```
auto lambda = [](int a) {
    return a;
};
sort(a.begin(), a.end(), ()[] { return x < y; })
```

- `SLS_All` (in configuration: `All`) Merge all lambdas fitting on a single line.

```
auto lambda = [](int a) {}
auto lambda2 = [](int a) { return a; };
```

AllowShortLoopsOnASingleLine (`Boolean`**)** clang-format 3.7

If `true`, `while (true) continue;` can be put on a single line.

AlwaysBreakAfterDefinitionReturnType (`DefinitionReturnTypeBreakingStyle`**)** clang-format 3.7

The function definition return type breaking style to use. This option is **deprecated** and is retained for backwards compatibility.

Possible values:

- `DRTBS_None` (in configuration: `None`) Break after return type automatically. `PenaltyReturnTypeOnItsOwnLine` is taken into account.
- `DRTBS_All` (in configuration: `All`) Always break after the return type.
- `DRTBS_TopLevel` (in configuration: `TopLevel`) Always break after the return types of top-level functions.

AlwaysBreakAfterReturnType (`ReturnTypeBreakingStyle`**)** clang-format 3.8

The function declaration return type breaking style to use.

Possible values:

- `RTBS_None` (in configuration: `None`) Break after return type automatically. `PenaltyReturnTypeOnItsOwnLine` is taken into account.

```
class A {
  int f() { return 0; };
};
int f();
int f() { return 1; }
```

- `RTBS_All` (in configuration: `All`) Always break after the return type.

```
class A {
  int
  f() {
    return 0;
  };
};
int
f();
int
f() {
  return 1;
}
```

- `RTBS_TopLevel` (in configuration: `TopLevel`) Always break after the return types of top-level functions.

```
class A {
  int f() { return 0; };
};
int
f();
int
f() {
  return 1;
}
```

- `RTBS_AllDefinitions` (in configuration: `AllDefinitions`) Always break after the return type of function definitions.

```
class A {
  int
  f() {
    return 0;
  };
};
int f();
int
f() {
  return 1;
}
```

- `RTBS_TopLevelDefinitions` (in configuration: `TopLevelDefinitions`) Always break after the return type of top-level definitions.

```
class A {
  int f() { return 0; };
};
int f();
int
f() {
  return 1;
}
```

AlwaysBreakBeforeMultilineStrings (`Boolean`) clang-format 3.4

If `true`, always break before multiline string literals.

This flag is mean to make cases where there are multiple multiline strings in a file look more consistent. Thus, it will only take effect if wrapping the string at that point leads to it being indented `ContinuationIndentWidth` spaces from the start of the line.

```
true:                                   false:
aaaa =                          vs.     aaaa = "bbbb"
    "bbbb"                                     "cccc";
    "cccc";
```

AlwaysBreakTemplateDeclarations (`BreakTemplateDeclarationsStyle`) clang-format 7

The template declaration breaking style to use.

Possible values:

- `BTDS_No` (in configuration: `No`) Do not force break before declaration. `PenaltyBreakTemplateDeclaration` is taken into account.

```
template <typename T> T foo() {
}
template <typename T> T foo(int aaaaaaaaaaaaaaaaaaaa,
                            int bbbbbbbbbbbbbbbbbbbb) {
}
```

- `BTDS_MultiLine` (in configuration: `MultiLine`) Force break after template declaration only when the following declaration spans multiple lines.

```
template <typename T> T foo() {
}
template <typename T>
T foo(int aaaaaaaaaaaaaaaaaaaa,
      int bbbbbbbbbbbbbbbbbbbb) {
}
```

- `BTDS_Yes` (in configuration: `Yes`) Always break after template declaration.

```
template <typename T>
T foo() {
}
template <typename T>
T foo(int aaaaaaaaaaaaaaaaaaaa,
      int bbbbbbbbbbbbbbbbbbbb) {
}
```

AttributeMacros (`List of Strings`) clang-format 12

A vector of strings that should be interpreted as attributes/qualifiers instead of identifiers. This can be useful for language extensions or static analyzer annotations.

For example:

```
x = (char *__capability)&y;
int function(void) __ununsed;
void only_writes_to_buffer(char *__output buffer);
```

In the .clang-format configuration file, this can be configured like:

```
AttributeMacros: ['__capability', '__output', '__ununsed']
```

BinPackArguments (`Boolean`) clang-format 3.7

If `false`, a function call's arguments will either be all on the same line or will have one line each.

```
true:
void f() {
  f(aaaaaaaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaaa,
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa);
}
```

```
false:
void f() {
  f(aaaaaaaaaaaaaaaaaaa,
    aaaaaaaaaaaaaaaaaaa,
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa);
}
```

BinPackParameters (`Boolean`) clang-format 3.7

If `false`, a function declaration's or function definition's parameters will either all be on the same line or will have one line each.

```
true:
void f(int aaaaaaaaaaaaaaaaaaa, int aaaaaaaaaaaaaaaaaaa,
       int aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa) {}

false:
void f(int aaaaaaaaaaaaaaaaaaa,
       int aaaaaaaaaaaaaaaaaaa,
       int aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa) {}
```

BitFieldColonSpacing (`BitFieldColonSpacingStyle`) clang-format 12

The BitFieldColonSpacingStyle to use for bitfields.

Possible values:

- `BFCS_Both` (in configuration: `Both`) Add one space on each side of the `:`

  ```
  unsigned bf : 2;
  ```

- `BFCS_None` (in configuration: `None`) Add no space around the `:` (except when needed for `AlignConsecutiveBitFields`).

  ```
  unsigned bf:2;
  ```

- `BFCS_Before` (in configuration: `Before`) Add space before the `:` only

  ```
  unsigned bf :2;
  ```

- `BFCS_After` (in configuration: `After`) Add space after the `:` only (space may be added before if needed for `AlignConsecutiveBitFields`).

  ```
  unsigned bf: 2;
  ```

BraceWrapping (`BraceWrappingFlags`) clang-format 3.8

Control of individual brace wrapping cases.

If `BreakBeforeBraces` is set to `BS_Custom`, use this to specify how each individual brace case should be handled. Otherwise, this is ignored.

```
# Example of usage:
BreakBeforeBraces: Custom
BraceWrapping:
  AfterEnum: true
  AfterStruct: false
  SplitEmptyFunction: false
```

Nested configuration flags:

Precise control over the wrapping of braces.

```
# Should be declared this way:
BreakBeforeBraces: Custom
BraceWrapping:
    AfterClass: true
```

- `bool AfterCaseLabel` Wrap case labels.

```
false:                                  vs.     true:
switch (foo) {                                  switch (foo) {
  case 1: {                                       case 1:
    bar();                                        {
    break;                                          bar();
  }                                                 break;
  default: {                                      }
    plop();                                       default:
  }                                               {
}                                                   plop();
                                                  }
                                                }
```

- `bool AfterClass` Wrap class definitions.

```
true:
class foo {};

false:
class foo
{};
```

- `BraceWrappingAfterControlStatementStyle AfterControlStatement` Wrap control statements (`if`/`for`/`while`/`switch`/..).

  Possible values:

  - `BWACS_Never` (in configuration: `Never`) Never wrap braces after a control statement.

```
if (foo()) {
} else {
}
for (int i = 0; i < 10; ++i) {
}
```

  - `BWACS_MultiLine` (in configuration: `MultiLine`) Only wrap braces after a multi-line control statement.

```
if (foo && bar &&
    baz)
{
  quux();
}
while (foo || bar) {
}
```

  - `BWACS_Always` (in configuration: `Always`) Always wrap braces after a control statement.

```
if (foo())
{
} else
{}
for (int i = 0; i < 10; ++i)
{}
```

- `bool AfterEnum` Wrap enum definitions.

```
true:
enum X : int
{
  B
};

false:
enum X : int { B };
```

- `bool AfterFunction` Wrap function definitions.

```
true:
void foo()
{
  bar();
  bar2();
}

false:
void foo() {
  bar();
  bar2();
}
```

- `bool AfterNamespace` Wrap namespace definitions.

```
true:
namespace
{
int foo();
int bar();
}

false:
namespace {
int foo();
int bar();
}
```

- `bool AfterObjCDeclaration` Wrap ObjC definitions (interfaces, implementations…). @autoreleasepool and @synchronized blocks are wrapped according to *AfterControlStatement* flag.

- `bool AfterStruct` Wrap struct definitions.

```
true:
struct foo
{
  int x;
};

false:
struct foo {
  int x;
};
```

- `bool AfterUnion` Wrap union definitions.

```
true:
union foo
{
  int x;
}

false:
union foo {
  int x;
}
```

- `bool AfterExternBlock` Wrap extern blocks.

```
true:
extern "C"
{
```

```
  int foo();
}

false:
extern "C" {
int foo();
}
```

- `bool BeforeCatch` Wrap before `catch`.

```
true:
try {
  foo();
}
catch () {
}

false:
try {
  foo();
} catch () {
}
```

- `bool BeforeElse` Wrap before `else`.

```
true:
if (foo()) {
}
else {
}

false:
if (foo()) {
} else {
}
```

- `bool BeforeLambdaBody` Wrap lambda block.

```
true:
connect(
  []()
  {
    foo();
    bar();
  });

false:
connect([]() {
  foo();
  bar();
});
```

- `bool BeforeWhile` Wrap before `while`.

```
true:
do {
  foo();
}
while (1);

false:
do {
```

```
  foo();
} while (1);
```

- `bool IndentBraces` Indent the wrapped braces themselves.
- `bool SplitEmptyFunction` If `false`, empty function body can be put on a single line. This option is used only if the opening brace of the function has already been wrapped, i.e. the *AfterFunction* brace wrapping mode is set, and the function could/should not be put on a single line (as per *AllowShortFunctionsOnASingleLine* and constructor formatting options).

```
false:          true:
int f()   vs.   int f()
{}              {
                }
```

- `bool SplitEmptyRecord` If `false`, empty record (e.g. class, struct or union) body can be put on a single line. This option is used only if the opening brace of the record has already been wrapped, i.e. the *AfterClass* (for classes) brace wrapping mode is set.

```
false:           true:
class Foo   vs.  class Foo
{}               {
                 }
```

- `bool SplitEmptyNamespace` If `false`, empty namespace body can be put on a single line. This option is used only if the opening brace of the namespace has already been wrapped, i.e. the *AfterNamespace* brace wrapping mode is set.

```
false:               true:
namespace Foo   vs.  namespace Foo
{}                   {
                     }
```

BreakAfterJavaFieldAnnotations (`Boolean`) clang-format 3.8

Break after each annotation on a field in Java files.

```
true:                                  false:
@Partial                       vs.     @Partial @Mock DataLoad loader;
@Mock
DataLoad loader;
```

BreakBeforeBinaryOperators (`BinaryOperatorStyle`) clang-format 3.6

The way to wrap binary operators.

Possible values:

- `BOS_None` (in configuration: `None`) Break after operators.

```
LooooooooooongType loooooooooooooooooooooongVariable =
    someLooooooooooooooooongFunction();

bool value = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa +
                 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa ==
             aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa &&
         aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa >
             ccccccccccccccccccccccccccccccccccccccccc;
```

- `BOS_NonAssignment` (in configuration: `NonAssignment`) Break before operators that aren't assignments.

```
LooooooooooongType loooooooooooooooooooooongVariable =
    someLooooooooooooooooongFunction();

bool value = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                 + aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
             == aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

```
                && aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                       > ccccccccccccccccccccccccccccccccccccccc;
```

- `BOS_All` (in configuration: `All`) Break before operators.

```
LooooooooooongType loooooooooooooooooooongVariable
    = someLooooooooooooooooongFunction();

bool value = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                     + aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                 == aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                && aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
                       > ccccccccccccccccccccccccccccccccccccccc;
```

BreakBeforeBraces (`BraceBreakingStyle`) clang-format 3.7

The brace breaking style to use.

Possible values:

- `BS_Attach` (in configuration: `Attach`) Always attach braces to surrounding context.

```
namespace N {
enum E {
  E1,
  E2,
};

class C {
public:
  C();
};

bool baz(int i) {
  try {
    do {
      switch (i) {
      case 1: {
        foobar();
        break;
      }
      default: {
        break;
      }
      }
    } while (--i);
    return true;
  } catch (...) {
    handleError();
    return false;
  }
}

void foo(bool b) {
  if (b) {
    baz(2);
  } else {
    baz(5);
  }
}

void bar() { foo(true); }
} // namespace N
```

- `BS_Linux` (in configuration: `Linux`) Like `Attach`, but break before braces on function, namespace and class definitions.

```
namespace N
{
enum E {
  E1,
  E2,
};

class C
{
public:
  C();
};

bool baz(int i)
{
  try {
    do {
      switch (i) {
      case 1: {
        foobar();
        break;
      }
      default: {
        break;
      }
      }
    } while (--i);
    return true;
  } catch (...) {
    handleError();
    return false;
  }
}

void foo(bool b)
{
  if (b) {
    baz(2);
  } else {
    baz(5);
  }
}

void bar() { foo(true); }
} // namespace N
```

- `BS_Mozilla` (in configuration: `Mozilla`) Like `Attach`, but break before braces on enum, function, and record definitions.

```
namespace N {
enum E
{
  E1,
  E2,
};

class C
{
```

```
public:
  C();
};

bool baz(int i)
{
  try {
    do {
      switch (i) {
      case 1: {
        foobar();
        break;
      }
      default: {
        break;
      }
      }
    } while (--i);
    return true;
  } catch (...) {
    handleError();
    return false;
  }
}

void foo(bool b)
{
  if (b) {
    baz(2);
  } else {
    baz(5);
  }
}

void bar() { foo(true); }
} // namespace N
```

- `BS_Stroustrup` (in configuration: `Stroustrup`) Like `Attach`, but break before function definitions, `catch`, and `else`.

```
namespace N {
enum E {
  E1,
  E2,
};

class C {
public:
  C();
};

bool baz(int i)
{
  try {
    do {
      switch (i) {
      case 1: {
        foobar();
        break;
      }
```

```
      default: {
        break;
      }
      }
    } while (--i);
    return true;
  }
  catch (...) {
    handleError();
    return false;
  }
}

void foo(bool b)
{
  if (b) {
    baz(2);
  }
  else {
    baz(5);
  }
}

void bar() { foo(true); }
} // namespace N
```

- `BS_Allman` (in configuration: `Allman`) Always break before braces.

```
namespace N
{
enum E
{
  E1,
  E2,
};

class C
{
public:
  C();
};

bool baz(int i)
{
  try
  {
    do
    {
      switch (i)
      {
      case 1:
      {
        foobar();
        break;
      }
      default:
      {
        break;
      }
      }
```

```
    } while (--i);
    return true;
  }
  catch (...)
  {
    handleError();
    return false;
  }
}

void foo(bool b)
{
  if (b)
  {
    baz(2);
  }
  else
  {
    baz(5);
  }
}

void bar() { foo(true); }
} // namespace N
```

- `BS_Whitesmiths` (in configuration: `Whitesmiths`) Like `Allman` but always indent braces and line up code with braces.

```
namespace N
  {
enum E
  {
  E1,
  E2,
  };

class C
  {
public:
  C();
  };

bool baz(int i)
  {
  try
    {
    do
      {
      switch (i)
        {
        case 1:
        {
        foobar();
        break;
        }
        default:
        {
        break;
        }
        }
```

```
    } while (--i);
  return true;
  }
catch (...)
  {
  handleError();
  return false;
  }
}

void foo(bool b)
  {
  if (b)
    {
    baz(2);
    }
  else
    {
    baz(5);
    }
  }

void bar() { foo(true); }
  } // namespace N
```

- `BS_GNU` (in configuration: `GNU`) Always break before braces and add an extra level of indentation to braces of control statements, not to those of class, function or other definitions.

```
namespace N
{
enum E
{
  E1,
  E2,
};

class C
{
public:
  C();
};

bool baz(int i)
{
  try
    {
      do
        {
          switch (i)
            {
            case 1:
              {
                foobar();
                break;
              }
            default:
              {
                break;
              }
            }
```

```
        }
      while (--i);
      return true;
    }
  catch (...)
    {
      handleError();
      return false;
    }
}

void foo(bool b)
{
  if (b)
    {
      baz(2);
    }
  else
    {
      baz(5);
    }
}

void bar() { foo(true); }
} // namespace N
```

- `BS_WebKit` (in configuration: `WebKit`) Like `Attach`, but break before functions.

```
namespace N {
enum E {
  E1,
  E2,
};

class C {
public:
  C();
};

bool baz(int i)
{
  try {
    do {
      switch (i) {
      case 1: {
        foobar();
        break;
      }
      default: {
        break;
      }
      }
    } while (--i);
    return true;
  } catch (...) {
    handleError();
    return false;
  }
}
```

```
void foo(bool b)
{
  if (b) {
    baz(2);
  } else {
    baz(5);
  }
}

void bar() { foo(true); }
} // namespace N
```

- `BS_Custom` (in configuration: `Custom`) Configure each individual brace in *BraceWrapping*.

BreakBeforeConceptDeclarations (`BreakBeforeConceptDeclarationsStyle`) clang-format 12

The concept declaration style to use.

Possible values:

- `BBCDS_Never` (in configuration: `Never`) Keep the template declaration line together with `concept`.

  ```
  template <typename T> concept C = ...;
  ```

- `BBCDS_Allowed` (in configuration: `Allowed`) Breaking between template declaration and `concept` is allowed. The actual behavior depends on the content and line breaking rules and penalities.

- `BBCDS_Always` (in configuration: `Always`) Always break before `concept`, putting it in the line after the template declaration.

  ```
  template <typename T>
  concept C = ...;
  ```

BreakBeforeTernaryOperators (`Boolean`) clang-format 3.7

If `true`, ternary operators will be placed after line breaks.

```
true:
veryVeryVeryVeryVeryVeryVeryVeryVeryVeryVeryLongDescription
    ? firstValue
    : SecondValueVeryVeryVeryVeryLong;

false:
veryVeryVeryVeryVeryVeryVeryVeryVeryVeryVeryLongDescription ?
    firstValue :
    SecondValueVeryVeryVeryVeryLong;
```

BreakConstructorInitializers (`BreakConstructorInitializersStyle`) clang-format 5

The break constructor initializers style to use.

Possible values:

- `BCIS_BeforeColon` (in configuration: `BeforeColon`) Break constructor initializers before the colon and after the commas.

  ```
  Constructor()
      : initializer1(),
        initializer2()
  ```

- `BCIS_BeforeComma` (in configuration: `BeforeComma`) Break constructor initializers before the colon and commas, and align the commas with the colon.

  ```
  Constructor()
      : initializer1()
      , initializer2()
  ```

- `BCIS_AfterColon` (in configuration: `AfterColon`) Break constructor initializers after the colon and commas.

```
    Constructor() :
        initializer1(),
        initializer2()
```

BreakInheritanceList (`BreakInheritanceListStyle`) clang-format 7

The inheritance list style to use.

Possible values:

- `BILS_BeforeColon` (in configuration: `BeforeColon`) Break inheritance list before the colon and after the commas.

```
class Foo
    : Base1,
      Base2
{};
```

- `BILS_BeforeComma` (in configuration: `BeforeComma`) Break inheritance list before the colon and commas, and align the commas with the colon.

```
class Foo
    : Base1
    , Base2
{};
```

- `BILS_AfterColon` (in configuration: `AfterColon`) Break inheritance list after the colon and commas.

```
class Foo :
    Base1,
    Base2
{};
```

- `BILS_AfterComma` (in configuration: `AfterComma`) Break inheritance list only after the commas.

```
class Foo : Base1,
            Base2
{};
```

BreakStringLiterals (`Boolean`) clang-format 3.9

Allow breaking string literals when formatting.

```
true:
const char* x = "veryVeryVeryVeryVeryVe"
                "ryVeryVeryVeryVeryVery"
                "VeryLongString";

false:
const char* x =
  "veryVeryVeryVeryVeryVeryVeryVeryVeryVeryVeryVeryLongString";
```

ColumnLimit (`Unsigned`) clang-format 3.7

The column limit.

A column limit of `0` means that there is no column limit. In this case, clang-format will respect the input's line breaking decisions within statements unless they contradict other rules.

CommentPragmas (`String`) clang-format 3.7

A regular expression that describes comments with special meaning, which should not be split into lines or otherwise changed.

```
// CommentPragmas: '^ FOOBAR pragma:'
// Will leave the following line unaffected
#include <vector> // FOOBAR pragma: keep
```

CompactNamespaces (`Boolean`) clang-format 5

If `true`, consecutive namespace declarations will be on the same line. If `false`, each namespace is declared on a new line.

```
true:
namespace Foo { namespace Bar {
}}

false:
namespace Foo {
namespace Bar {
}
}
```

If it does not fit on a single line, the overflowing namespaces get wrapped:

```
namespace Foo { namespace Bar {
namespace Extra {
}}}
```

ConstructorInitializerAllOnOneLineOrOnePerLine (`Boolean`) clang-format 3.7

This option is **deprecated**. See `CurrentLine` of `PackConstructorInitializers`.

ConstructorInitializerIndentWidth (`Unsigned`) clang-format 3.7

The number of characters to use for indentation of constructor initializer lists as well as inheritance lists.

ContinuationIndentWidth (`Unsigned`) clang-format 3.7

Indent width for line continuations.

```
ContinuationIndentWidth: 2

int i =         //  VeryVeryVeryVeryVeryLongComment
  longFunction( // Again a long comment
    arg);
```

Cpp11BracedListStyle (`Boolean`) clang-format 3.4

If `true`, format braced lists as best suited for C++11 braced lists.

Important differences: - No spaces inside the braced list. - No line break before the closing brace. - Indentation with the continuation indent, not with the block indent.

Fundamentally, C++11 braced lists are formatted exactly like function calls would be formatted in their place. If the braced list follows a name (e.g. a type or variable name), clang-format formats as if the `{}` were the parentheses of a function call with that name. If there is no name, a zero-length name is assumed.

```
true:                                   false:
vector<int> x{1, 2, 3, 4};     vs.      vector<int> x{ 1, 2, 3, 4 };
vector<T> x{{}, {}, {}, {}};            vector<T> x{ {}, {}, {}, {} };
f(MyMap[{composite, key}]);             f(MyMap[{ composite, key }]);
new int[3]{1, 2, 3};                    new int[3]{ 1, 2, 3 };
```

DeriveLineEnding (`Boolean`) clang-format 10

Analyze the formatted file for the most used line ending (`\r\n` or `\n`). `UseCRLF` is only used as a fallback if none can be derived.

DerivePointerAlignment (`Boolean`) clang-format 3.7

If `true`, analyze the formatted file for the most common alignment of `&` and `*`. Pointer and reference alignment styles are going to be updated according to the preferences found in the file. `PointerAlignment` is then used only as fallback.

DisableFormat (`Boolean`) clang-format 3.7

Disables formatting completely.

EmptyLineAfterAccessModifier (`EmptyLineAfterAccessModifierStyle`) clang-format 13

Defines when to put an empty line after access modifiers. `EmptyLineBeforeAccessModifier` configuration handles the number of empty lines between two access modifiers.

Possible values:

- `ELAAMS_Never` (in configuration: `Never`) Remove all empty lines after access modifiers.

```
struct foo {
private:
  int i;
protected:
  int j;
  /* comment */
public:
  foo() {}
private:
protected:
};
```

- `ELAAMS_Leave` (in configuration: `Leave`) Keep existing empty lines after access modifiers. MaxEmptyLinesToKeep is applied instead.
- `ELAAMS_Always` (in configuration: `Always`) Always add empty line after access modifiers if there are none. MaxEmptyLinesToKeep is applied also.

```
struct foo {
private:

  int i;
protected:

  int j;
  /* comment */
public:

  foo() {}
private:

protected:

};
```

EmptyLineBeforeAccessModifier (`EmptyLineBeforeAccessModifierStyle`) clang-format 12

Defines in which cases to put empty line before access modifiers.

Possible values:

- `ELBAMS_Never` (in configuration: `Never`) Remove all empty lines before access modifiers.

```
struct foo {
private:
  int i;
protected:
  int j;
  /* comment */
public:
  foo() {}
private:
protected:
};
```

- `ELBAMS_Leave` (in configuration: `Leave`) Keep existing empty lines before access modifiers.
- `ELBAMS_LogicalBlock` (in configuration: `LogicalBlock`) Add empty line only when access modifier starts a new logical block. Logical block is a group of one or more member fields or functions.

```
struct foo {
private:
  int i;
```

```
protected:
  int j;
  /* comment */
public:
  foo() {}

private:
protected:
};
```

- `ELBAMS_Always` (in configuration: `Always`) Always add empty line before access modifiers unless access modifier is at the start of struct or class definition.

```
struct foo {
private:
  int i;

protected:
  int j;
  /* comment */

public:
  foo() {}

private:

protected:
};
```

ExperimentalAutoDetectBinPacking (`Boolean`) clang-format 3.7

If `true`, clang-format detects whether function calls and definitions are formatted with one parameter per line.

Each call can be bin-packed, one-per-line or inconclusive. If it is inconclusive, e.g. completely on one line, but a decision needs to be made, clang-format analyzes whether there are other bin-packed cases in the input file and act accordingly.

NOTE: This is an experimental flag, that might go away or be renamed. Do not use this in config files, etc. Use at your own risk.

FixNamespaceComments (`Boolean`) clang-format 5

If `true`, clang-format adds missing namespace end comments for short namespaces and fixes invalid existing ones. Short ones are controlled by “ShortNamespaceLines”.

```
true:                                   false:
namespace a {                  vs.      namespace a {
foo();                                  foo();
bar();                                  bar();
} // namespace a                        }
```

ForEachMacros (`List of Strings`) clang-format 3.7

A vector of macros that should be interpreted as foreach loops instead of as function calls.

These are expected to be macros of the form:

```
FOREACH(<variable-declaration>, ...)
  <loop-body>
```

In the .clang-format configuration file, this can be configured like:

```
ForEachMacros: ['RANGES_FOR', 'FOREACH']
```

For example: BOOST_FOREACH.

IfMacros (`List of Strings`) clang-format 13

A vector of macros that should be interpreted as conditionals instead of as function calls.

These are expected to be macros of the form:

```
IF(...)
  <conditional-body>
else IF(...)
  <conditional-body>
```

In the .clang-format configuration file, this can be configured like:

```
IfMacros: ['IF']
```

For example: KJ_IF_MAYBE

IncludeBlocks (`IncludeBlocksStyle`) clang-format 7

Dependent on the value, multiple `#include` blocks can be sorted as one and divided based on category.

Possible values:

- `IBS_Preserve` (in configuration: `Preserve`) Sort each `#include` block separately.

```
#include "b.h"                 into      #include "b.h"

#include <lib/main.h>                    #include "a.h"
#include "a.h"                           #include <lib/main.h>
```

- `IBS_Merge` (in configuration: `Merge`) Merge multiple `#include` blocks together and sort as one.

```
#include "b.h"                 into      #include "a.h"
                                         #include "b.h"
#include <lib/main.h>                    #include <lib/main.h>
#include "a.h"
```

- `IBS_Regroup` (in configuration: `Regroup`) Merge multiple `#include` blocks together and sort as one. Then split into groups based on category priority. See `IncludeCategories`.

```
#include "b.h"                 into      #include "a.h"
                                         #include "b.h"
#include <lib/main.h>
#include "a.h"                           #include <lib/main.h>
```

IncludeCategories (`List of IncludeCategories`) clang-format 7

Regular expressions denoting the different `#include` categories used for ordering `#includes`.

POSIX extended regular expressions are supported.

These regular expressions are matched against the filename of an include (including the <> or “”) in order. The value belonging to the first matching regular expression is assigned and `#includes` are sorted first according to increasing category number and then alphabetically within each category.

If none of the regular expressions match, INT_MAX is assigned as category. The main header for a source file automatically gets category 0. so that it is generally kept at the beginning of the `#includes` (https://llvm.org/docs/CodingStandards.html#include-style). However, you can also assign negative priorities if you have certain headers that always need to be first.

There is a third and optional field `SortPriority` which can used while `IncludeBlocks = IBS_Regroup` to define the priority in which `#includes` should be ordered. The value of `Priority` defines the order of `#include blocks` and also allows the grouping of `#includes` of different priority. `SortPriority` is set to the value of `Priority` as default if it is not assigned.

Each regular expression can be marked as case sensitive with the field `CaseSensitive`, per default it is not.

To configure this in the .clang-format file, use:

```
IncludeCategories:
  - Regex:           '^"(llvm|llvm-c|clang|clang-c)/'
    Priority:        2
    SortPriority:    2
    CaseSensitive:   true
```

```
  - Regex:           '^((<|")(gtest|gmock|isl|json)/)'
    Priority:        3
  - Regex:           '<[[:alnum:].]+>'
    Priority:        4
  - Regex:           '.*'
    Priority:        1
    SortPriority:    0
```

IncludeIsMainRegex (`String`) clang-format 7

Specify a regular expression of suffixes that are allowed in the file-to-main-include mapping.

When guessing whether a #include is the “main” include (to assign category 0, see above), use this regex of allowed suffixes to the header stem. A partial match is done, so that: - “” means “arbitrary suffix” - “$” means “no suffix”

For example, if configured to “(_test)?$”, then a header a.h would be seen as the “main” include in both a.cc and a_test.cc.

IncludeIsMainSourceRegex (`String`) clang-format 7

Specify a regular expression for files being formatted that are allowed to be considered “main” in the file-to-main-include mapping.

By default, clang-format considers files as “main” only when they end with: `.c`, `.cc`, `.cpp`, `.c++`, `.cxx`, `.m` or `.mm` extensions. For these files a guessing of “main” include takes place (to assign category 0, see above). This config option allows for additional suffixes and extensions for files to be considered as “main”.

For example, if this option is configured to `(Impl.hpp)$`, then a file `ClassImpl.hpp` is considered “main” (in addition to `Class.c`, `Class.cc`, `Class.cpp` and so on) and “main include file” logic will be executed (with *IncludeIsMainRegex* setting also being respected in later phase). Without this option set, `ClassImpl.hpp` would not have the main include file put on top before any other include.

IndentAccessModifiers (`Boolean`) clang-format 13

Specify whether access modifiers should have their own indentation level.

When `false`, access modifiers are indented (or outdented) relative to the record members, respecting the `AccessModifierOffset`. Record members are indented one level below the record. When `true`, access modifiers get their own indentation level. As a consequence, record members are always indented 2 levels below the record, regardless of the access modifier presence. Value of the `AccessModifierOffset` is ignored.

```
false:                                  true:
class C {                       vs.     class C {
  class D {                                 class D {
    void bar();                                 void bar();
  protected:                                  protected:
    D();                                        D();
  };                                        };
public:                                   public:
  C();                                      C();
};                                      };
void foo() {                            void foo() {
  return 1;                               return 1;
}                                       }
```

IndentCaseBlocks (`Boolean`) clang-format 11

Indent case label blocks one level from the case label.

When `false`, the block following the case label uses the same indentation level as for the case label, treating the case label the same as an if-statement. When `true`, the block gets indented as a scope block.

```
false:                                  true:
switch (fool) {                 vs.     switch (fool) {
case 1: {                               case 1:
  bar();                                  {
} break;                                    bar();
```

```
default: {
  plop();
}
}
```

```
  }
  break;
default:
  {
    plop();
  }
}
```

IndentCaseLabels (`Boolean`) clang-format 3.3

Indent case labels one level from the switch statement.

When `false`, use the same indentation level as for the switch statement. Switch statement body is always indented one level more than case labels (except the first block following the case label, which itself indents the code - unless IndentCaseBlocks is enabled).

```
false:
switch (fool) {
case 1:
  bar();
  break;
default:
  plop();
}
```

vs.

```
true:
switch (fool) {
  case 1:
    bar();
    break;
  default:
    plop();
}
```

IndentExternBlock (`IndentExternBlockStyle`) clang-format 11

IndentExternBlockStyle is the type of indenting of extern blocks.

Possible values:

- `IEBS_AfterExternBlock` (in configuration: `AfterExternBlock`) Backwards compatible with AfterExternBlock's indenting.

```
IndentExternBlock: AfterExternBlock
BraceWrapping.AfterExternBlock: true
extern "C"
{
    void foo();
}

IndentExternBlock: AfterExternBlock
BraceWrapping.AfterExternBlock: false
extern "C" {
void foo();
}
```

- `IEBS_NoIndent` (in configuration: `NoIndent`) Does not indent extern blocks.

```
extern "C" {
void foo();
}
```

- `IEBS_Indent` (in configuration: `Indent`) Indents extern blocks.

```
extern "C" {
  void foo();
}
```

IndentGotoLabels (`Boolean`) clang-format 10

Indent goto labels.

When `false`, goto labels are flushed left.

```
true:
int f() {
  if (foo()) {
  label1:
```

vs.

```
false:
int f() {
  if (foo()) {
label1:
```

```
    bar();                                    bar();
  }                                         }
label2:                                   label2:
  return 1;                                 return 1;
}                                         }
```

IndentPPDirectives (`PPDirectiveIndentStyle`**)** clang-format 6

The preprocessor directive indenting style to use.

Possible values:

- `PPDIS_None` (in configuration: `None`) Does not indent any directives.

```
#if FOO
#if BAR
#include <foo>
#endif
#endif
```

- `PPDIS_AfterHash` (in configuration: `AfterHash`) Indents directives after the hash.

```
#if FOO
#  if BAR
#    include <foo>
#  endif
#endif
```

- `PPDIS_BeforeHash` (in configuration: `BeforeHash`) Indents directives before the hash.

```
#if FOO
  #if BAR
    #include <foo>
  #endif
#endif
```

IndentRequiresClause (`Boolean`**)** clang-format 15

Indent the requires clause in a template. This only applies when `RequiresClausePosition` is `OwnLine`, or `WithFollowing`.

In clang-format 12, 13 and 14 it was named `IndentRequires`.

```
true:
template <typename It>
  requires Iterator<It>
void sort(It begin, It end) {
  //....
}

false:
template <typename It>
requires Iterator<It>
void sort(It begin, It end) {
  //....
}
```

IndentWidth (`Unsigned`**)** clang-format 3.7

The number of columns to use for indentation.

```
IndentWidth: 3

void f() {
   someFunction();
   if (true, false) {
      f();
   }
}
```

IndentWrappedFunctionNames (`Boolean`) clang-format 3.7

Indent if a function definition or declaration is wrapped after the type.

```
true:
LooooooooooooooooooooooooooooooooooooooongReturnType
    LoooooooooooooooooooooooooooooongFunctionDeclaration();

false:
LooooooooooooooooooooooooooooooooooooooongReturnType
LoooooooooooooooooooooooooooooongFunctionDeclaration();
```

InsertBraces (`Boolean`) clang-format 15

Insert braces after control statements (`if`, `else`, `for`, `do`, and `while`) in C++ unless the control statements are inside macro definitions or the braces would enclose preprocessor directives.

> **Warning**
>
> Setting this option to *true* could lead to incorrect code formatting due to clang-format's lack of complete semantic information. As such, extra care should be taken to review code changes made by this option.

```
false:                                  true:

if (isa<FunctionDecl>(D))        vs.    if (isa<FunctionDecl>(D)) {
  handleFunctionDecl(D);                  handleFunctionDecl(D);
else if (isa<VarDecl>(D))                } else if (isa<VarDecl>(D)) {
  handleVarDecl(D);                       handleVarDecl(D);
else                                     } else {
  return;                                 return;
                                         }

while (i--)                      vs.    while (i--) {
  for (auto *A : D.attrs())                for (auto *A : D.attrs()) {
    handleAttr(A);                           handleAttr(A);
                                           }
                                         }

do                               vs.    do {
  --i;                                    --i;
while (i);                               } while (i);
```

InsertTrailingCommas (`TrailingCommaStyle`) clang-format 11

If set to `TCS_Wrapped` will insert trailing commas in container literals (arrays and objects) that wrap across multiple lines. It is currently only available for JavaScript and disabled by default `TCS_None`. `InsertTrailingCommas` cannot be used together with `BinPackArguments` as inserting the comma disables bin-packing.

```
TSC_Wrapped:
const someArray = [
aaaaaaaaaaaaaaaaaaaaaaaaa,
aaaaaaaaaaaaaaaaaaaaaaaaa,
aaaaaaaaaaaaaaaaaaaaaaaaa,
//                        ^ inserted
]
```

Possible values:

- `TCS_None` (in configuration: `None`) Do not insert trailing commas.
- `TCS_Wrapped` (in configuration: `Wrapped`) Insert trailing commas in container literals that were wrapped over multiple lines. Note that this is conceptually incompatible with bin-packing, because the trailing

comma is used as an indicator that a container should be formatted one-per-line (i.e. not bin-packed). So inserting a trailing comma counteracts bin-packing.

JavaImportGroups (`List of Strings`) clang-format 8

A vector of prefixes ordered by the desired groups for Java imports.

One group's prefix can be a subset of another - the longest prefix is always matched. Within a group, the imports are ordered lexicographically. Static imports are grouped separately and follow the same group rules. By default, static imports are placed before non-static imports, but this behavior is changed by another option, `SortJavaStaticImport`.

In the .clang-format configuration file, this can be configured like in the following yaml example. This will result in imports being formatted as in the Java example below.

```
JavaImportGroups: ['com.example', 'com', 'org']

import static com.example.function1;

import static com.test.function2;

import static org.example.function3;

import com.example.ClassA;
import com.example.Test;
import com.example.a.ClassB;

import com.test.ClassC;

import org.example.ClassD;
```

JavaScriptQuotes (`JavaScriptQuoteStyle`) clang-format 3.9

The JavaScriptQuoteStyle to use for JavaScript strings.

Possible values:

- `JSQS_Leave` (in configuration: `Leave`) Leave string quotes as they are.

  ```
  string1 = "foo";
  string2 = 'bar';
  ```

- `JSQS_Single` (in configuration: `Single`) Always use single quotes.

  ```
  string1 = 'foo';
  string2 = 'bar';
  ```

- `JSQS_Double` (in configuration: `Double`) Always use double quotes.

  ```
  string1 = "foo";
  string2 = "bar";
  ```

JavaScriptWrapImports (`Boolean`) clang-format 3.9

Whether to wrap JavaScript import/export statements.

```
true:
import {
    VeryLongImportsAreAnnoying,
    VeryLongImportsAreAnnoying,
    VeryLongImportsAreAnnoying,
} from 'some/module.js'

false:
import {VeryLongImportsAreAnnoying, VeryLongImportsAreAnnoying, VeryLongImportsAreAnnoying,} from "some/module.js"
```

KeepEmptyLinesAtTheStartOfBlocks (`Boolean`) clang-format 3.7

If true, the empty line at the start of blocks is kept.

```
true:                                   false:
if (foo) {                     vs.      if (foo) {
                                          bar();
```

```
  bar();                                  }
}
```

LambdaBodyIndentation (`LambdaBodyIndentationKind`) clang-format 13

The indentation style of lambda bodies. `Signature` (the default) causes the lambda body to be indented one additional level relative to the indentation level of the signature. `OuterScope` forces the lambda body to be indented one additional level relative to the parent scope containing the lambda signature. For callback-heavy code, it may improve readability to have the signature indented two levels and to use `OuterScope`. The KJ style guide requires `OuterScope`. KJ style guide

Possible values:

- `LBI_Signature` (in configuration: `Signature`) Align lambda body relative to the lambda signature. This is the default.

```
someMethod(
    [](SomeReallyLongLambdaSignatureArgument foo) {
      return;
    });
```

- `LBI_OuterScope` (in configuration: `OuterScope`) Align lambda body relative to the indentation level of the outer scope the lambda signature resides in.

```
someMethod(
    [](SomeReallyLongLambdaSignatureArgument foo) {
  return;
});
```

Language (`LanguageKind`) clang-format 3.5

Language, this format style is targeted at.

Possible values:

- `LK_None` (in configuration: `None`) Do not use.
- `LK_Cpp` (in configuration: `Cpp`) Should be used for C, C++.
- `LK_CSharp` (in configuration: `CSharp`) Should be used for C#.
- `LK_Java` (in configuration: `Java`) Should be used for Java.
- `LK_JavaScript` (in configuration: `JavaScript`) Should be used for JavaScript.
- `LK_Json` (in configuration: `Json`) Should be used for JSON.
- `LK_ObjC` (in configuration: `ObjC`) Should be used for Objective-C, Objective-C++.
- `LK_Proto` (in configuration: `Proto`) Should be used for Protocol Buffers (https://developers.google.com/protocol-buffers/).
- `LK_TableGen` (in configuration: `TableGen`) Should be used for TableGen code.
- `LK_TextProto` (in configuration: `TextProto`) Should be used for Protocol Buffer messages in text format (https://developers.google.com/protocol-buffers/).

MacroBlockBegin (`String`) clang-format 3.7

A regular expression matching macros that start a block.

```
# With:
MacroBlockBegin: "^NS_MAP_BEGIN|\
NS_TABLE_HEAD$"
MacroBlockEnd: "^\
NS_MAP_END|\
NS_TABLE_.*_END$"

NS_MAP_BEGIN
  foo();
NS_MAP_END
```

```
NS_TABLE_HEAD
  bar();
NS_TABLE_FOO_END

# Without:
NS_MAP_BEGIN
foo();
NS_MAP_END

NS_TABLE_HEAD
bar();
NS_TABLE_FOO_END
```

MacroBlockEnd (`String`) clang-format 3.7

A regular expression matching macros that end a block.

MaxEmptyLinesToKeep (`Unsigned`) clang-format 3.7

The maximum number of consecutive empty lines to keep.

```
MaxEmptyLinesToKeep: 1         vs.     MaxEmptyLinesToKeep: 0
int f() {                              int f() {
  int = 1;                                 int i = 1;
                                           i = foo();
  i = foo();                               return i;
                                       }
  return i;
}
```

NamespaceIndentation (`NamespaceIndentationKind`) clang-format 3.7

The indentation used for namespaces.

Possible values:

- `NI_None` (in configuration: `None`) Don't indent in namespaces.

  ```
  namespace out {
  int i;
  namespace in {
  int i;
  }
  }
  ```

- `NI_Inner` (in configuration: `Inner`) Indent only in inner namespaces (nested in other namespaces).

  ```
  namespace out {
  int i;
  namespace in {
    int i;
  }
  }
  ```

- `NI_All` (in configuration: `All`) Indent in all namespaces.

  ```
  namespace out {
    int i;
    namespace in {
      int i;
    }
  }
  ```

NamespaceMacros (`List of Strings`) clang-format 9

A vector of macros which are used to open namespace blocks.

These are expected to be macros of the form:

```
NAMESPACE(<namespace-name>, ...) {
  <namespace-content>
}
```

For example: TESTSUITE

ObjCBinPackProtocolList (`BinPackStyle`**)** clang-format 7

Controls bin-packing Objective-C protocol conformance list items into as few lines as possible when they go over `ColumnLimit`.

If `Auto` (the default), delegates to the value in `BinPackParameters`. If that is `true`, bin-packs Objective-C protocol conformance list items into as few lines as possible whenever they go over `ColumnLimit`.

If `Always`, always bin-packs Objective-C protocol conformance list items into as few lines as possible whenever they go over `ColumnLimit`.

If `Never`, lays out Objective-C protocol conformance list items onto individual lines whenever they go over `ColumnLimit`.

```
Always (or Auto, if BinPackParameters=true):
@interface ccccccccccccc () <
    ccccccccccccc, ccccccccccccc,
    ccccccccccccc, ccccccccccccc> {
}

Never (or Auto, if BinPackParameters=false):
@interface ddddddddddddd () <
    ddddddddddddd,
    ddddddddddddd,
    ddddddddddddd,
    ddddddddddddd> {
}
```

Possible values:

- `BPS_Auto` (in configuration: `Auto`) Automatically determine parameter bin-packing behavior.
- `BPS_Always` (in configuration: `Always`) Always bin-pack parameters.
- `BPS_Never` (in configuration: `Never`) Never bin-pack parameters.

ObjCBlockIndentWidth (`Unsigned`**)** clang-format 3.7

The number of characters to use for indentation of ObjC blocks.

```
ObjCBlockIndentWidth: 4

[operation setCompletionBlock:^{
    [self onOperationDone];
}];
```

ObjCBreakBeforeNestedBlockParam (`Boolean`**)** clang-format 11

Break parameters list into lines when there is nested block parameters in a function call.

```
false:
 - (void)_aMethod
 {
     [self.test1 t:self w:self callback:^(typeof(self) self, NSNumber
     *u, NSNumber *v) {
         u = c;
     }]
 }
 true:
 - (void)_aMethod
 {
    [self.test1 t:self
```

```
                  w:self
            callback:^(typeof(self) self, NSNumber *u, NSNumber *v) {
                u = c;
             }]
    }
```

ObjCSpaceAfterProperty (`Boolean`**)** clang-format 3.7

Add a space after `@property` in Objective-C, i.e. use `@property (readonly)` instead of `@property(readonly)`.

ObjCSpaceBeforeProtocolList (`Boolean`**)** clang-format 3.7

Add a space in front of an Objective-C protocol list, i.e. use `Foo <Protocol>` instead of `Foo<Protocol>`.

PPIndentWidth (`Integer`**)** clang-format 13

The number of columns to use for indentation of preprocessor statements. When set to -1 (default) `IndentWidth` is used also for preprocessor statements.

```
PPIndentWidth: 1

#ifdef __linux__
# define FOO
#else
# define BAR
#endif
```

PackConstructorInitializers (`PackConstructorInitializersStyle`**)** clang-format 14

The pack constructor initializers style to use.

Possible values:

- `PCIS_Never` (in configuration: `Never`) Always put each constructor initializer on its own line.

```
Constructor()
    : a(),
      b()
```

- `PCIS_BinPack` (in configuration: `BinPack`) Bin-pack constructor initializers.

```
Constructor()
    : aaaaaaaaaaaaaaaaaaa(), bbbbbbbbbbbbbbbbbbb(),
      ccccccccccccccccccc()
```

- `PCIS_CurrentLine` (in configuration: `CurrentLine`) Put all constructor initializers on the current line if they fit. Otherwise, put each one on its own line.

```
Constructor() : a(), b()

Constructor()
    : aaaaaaaaaaaaaaaaaaa(),
      bbbbbbbbbbbbbbbbbbb(),
      ddddddddddddd()
```

- `PCIS_NextLine` (in configuration: `NextLine`) Same as `PCIS_CurrentLine` except that if all constructor initializers do not fit on the current line, try to fit them on the next line.

```
Constructor() : a(), b()

Constructor()
    : aaaaaaaaaaaaaaaaaaa(), bbbbbbbbbbbbbbbbbbb(), ddddddddddddd()

Constructor()
    : aaaaaaaaaaaaaaaaaaa(),
      bbbbbbbbbbbbbbbbbbb(),
      ccccccccccccccccccc()
```

PenaltyBreakAssignment (`Unsigned`**)** clang-format 5

The penalty for breaking around an assignment operator.

PenaltyBreakBeforeFirstCallParameter (`Unsigned`) clang-format 3.7

The penalty for breaking a function call after `call(`.

PenaltyBreakComment (`Unsigned`) clang-format 3.7

The penalty for each line break introduced inside a comment.

PenaltyBreakFirstLessLess (`Unsigned`) clang-format 3.7

The penalty for breaking before the first <<.

PenaltyBreakOpenParenthesis (`Unsigned`) clang-format 14

The penalty for breaking after `(`.

PenaltyBreakString (`Unsigned`) clang-format 3.7

The penalty for each line break introduced inside a string literal.

PenaltyBreakTemplateDeclaration (`Unsigned`) clang-format 7

The penalty for breaking after template declaration.

PenaltyExcessCharacter (`Unsigned`) clang-format 3.7

The penalty for each character outside of the column limit.

PenaltyIndentedWhitespace (`Unsigned`) clang-format 12

Penalty for each character of whitespace indentation (counted relative to leading non-whitespace column).

PenaltyReturnTypeOnItsOwnLine (`Unsigned`) clang-format 3.7

Penalty for putting the return type of a function onto its own line.

PointerAlignment (`PointerAlignmentStyle`) clang-format 3.7

Pointer and reference alignment style.

Possible values:

- `PAS_Left` (in configuration: `Left`) Align pointer to the left.

```
int* a;
```

- `PAS_Right` (in configuration: `Right`) Align pointer to the right.

```
int *a;
```

- `PAS_Middle` (in configuration: `Middle`) Align pointer in the middle.

```
int * a;
```

QualifierAlignment (`QualifierAlignmentStyle`) clang-format 14

Different ways to arrange specifiers and qualifiers (e.g. const/volatile).

Warning

Setting `QualifierAlignment` to something other than *Leave*, COULD lead to incorrect code formatting due to incorrect decisions made due to clang-formats lack of complete semantic information. As such extra care should be taken to review code changes made by the use of this option.

Possible values:

- `QAS_Leave` (in configuration: `Leave`) Don't change specifiers/qualifiers to either Left or Right alignment (default).

```
int const a;
const int *a;
```

- `QAS_Left` (in configuration: `Left`) Change specifiers/qualifiers to be left-aligned.

```
const int a;
const int *a;
```

- `QAS_Right` (in configuration: `Right`) Change specifiers/qualifiers to be right-aligned.

```
int const a;
int const *a;
```

- `QAS_Custom` (in configuration: `Custom`) Change specifiers/qualifiers to be aligned based on `QualifierOrder`. With:

```
QualifierOrder: ['inline', 'static' , 'type', 'const']

int const a;
int const *a;
```

QualifierOrder (`List of Strings`) clang-format 14

The order in which the qualifiers appear. Order is an array that can contain any of the following:

- const
- inline
- static
- constexpr
- volatile
- restrict
- type

Note: it MUST contain 'type'. Items to the left of 'type' will be placed to the left of the type and aligned in the order supplied. Items to the right of 'type' will be placed to the right of the type and aligned in the order supplied.

```
QualifierOrder: ['inline', 'static', 'type', 'const', 'volatile' ]
```

RawStringFormats (`List of RawStringFormats`) clang-format 6

Defines hints for detecting supported languages code blocks in raw strings.

A raw string with a matching delimiter or a matching enclosing function name will be reformatted assuming the specified language based on the style for that language defined in the .clang-format file. If no style has been defined in the .clang-format file for the specific language, a predefined style given by 'BasedOnStyle' is used. If 'BasedOnStyle' is not found, the formatting is based on llvm style. A matching delimiter takes precedence over a matching enclosing function name for determining the language of the raw string contents.

If a canonical delimiter is specified, occurrences of other delimiters for the same language will be updated to the canonical if possible.

There should be at most one specification per language and each delimiter and enclosing function should not occur in multiple specifications.

To configure this in the .clang-format file, use:

```
RawStringFormats:
  - Language: TextProto
      Delimiters:
        - 'pb'
        - 'proto'
      EnclosingFunctions:
        - 'PARSE_TEXT_PROTO'
      BasedOnStyle: google
  - Language: Cpp
      Delimiters:
        - 'cc'
        - 'cpp'
      BasedOnStyle: llvm
      CanonicalDelimiter: 'cc'
```

ReferenceAlignment (`ReferenceAlignmentStyle`) clang-format 13

Reference alignment style (overrides `PointerAlignment` for references).

Possible values:

- `RAS_Pointer` (in configuration: `Pointer`) Align reference like `PointerAlignment`.
- `RAS_Left` (in configuration: `Left`) Align reference to the left.

  ```
  int& a;
  ```

- `RAS_Right` (in configuration: `Right`) Align reference to the right.

  ```
  int &a;
  ```

- `RAS_Middle` (in configuration: `Middle`) Align reference in the middle.

  ```
  int & a;
  ```

ReflowComments (`Boolean`) clang-format 4

If `true`, clang-format will attempt to re-flow comments.

```
false:
// veryVeryVeryVeryVeryVeryVeryVeryVeryVeryLongComment with plenty of information
/* second veryVeryVeryVeryVeryVeryVeryVeryVeryVeryLongComment with plenty of information */

true:
// veryVeryVeryVeryVeryVeryVeryVeryVeryVeryLongComment with plenty of
// information
/* second veryVeryVeryVeryVeryVeryVeryVeryVeryVeryLongComment with plenty of
 * information */
```

RemoveBracesLLVM (`Boolean`) clang-format 14

Remove optional braces of control statements (`if`, `else`, `for`, and `while`) in C++ according to the LLVM coding style.

> **Warning**
>
> This option will be renamed and expanded to support other styles.

> **Warning**
>
> Setting this option to *true* could lead to incorrect code formatting due to clang-format's lack of complete semantic information. As such, extra care should be taken to review code changes made by this option.

```
false:                                  true:

if (isa<FunctionDecl>(D)) {        vs.  if (isa<FunctionDecl>(D))
  handleFunctionDecl(D);                  handleFunctionDecl(D);
} else if (isa<VarDecl>(D)) {           else if (isa<VarDecl>(D))
  handleVarDecl(D);                       handleVarDecl(D);
}

if (isa<VarDecl>(D)) {             vs.  if (isa<VarDecl>(D)) {
  for (auto *A : D.attrs()) {             for (auto *A : D.attrs())
    if (shouldProcessAttr(A)) {             if (shouldProcessAttr(A))
      handleAttr(A);                          handleAttr(A);
    }                                     }
  }
}

if (isa<FunctionDecl>(D)) {        vs.  if (isa<FunctionDecl>(D))
  for (auto *A : D.attrs()) {             for (auto *A : D.attrs())
    handleAttr(A);                          handleAttr(A);
  }
```

```
}

if (auto *D = (T)(D)) {                vs.     if (auto *D = (T)(D)) {
  if (shouldProcess(D)) {                        if (shouldProcess(D))
    handleVarDecl(D);                              handleVarDecl(D);
  } else {                                       else
    markAsIgnored(D);                              markAsIgnored(D);
  }                                              }
}

if (a) {                               vs.     if (a)
  b();                                           b();
} else {                                       else if (c)
  if (c) {                                       d();
    d();                                       else
  } else {                                       e();
    e();
  }
}
```

RequiresClausePosition (`RequiresClausePositionStyle`) clang-format 15

The position of the `requires` clause.

Possible values:

- `RCPS_OwnLine` (in configuration: `OwnLine`) Always put the `requires` clause on its own line.

```
template <typename T>
requires C<T>
struct Foo {...

template <typename T>
requires C<T>
void bar(T t) {...

template <typename T>
void baz(T t)
requires C<T>
{...
```

- `RCPS_WithPreceding` (in configuration: `WithPreceding`) Try to put the clause together with the preceding part of a declaration. For class templates: stick to the template declaration. For function templates: stick to the template declaration. For function declaration followed by a requires clause: stick to the parameter list.

```
template <typename T> requires C<T>
struct Foo {...

template <typename T> requires C<T>
void bar(T t) {...

template <typename T>
void baz(T t) requires C<T>
{...
```

- `RCPS_WithFollowing` (in configuration: `WithFollowing`) Try to put the `requires` clause together with the class or function declaration.

```
template <typename T>
requires C<T> struct Foo {...

template <typename T>
requires C<T> void bar(T t) {...
```

```
template <typename T>
void baz(T t)
requires C<T> {...
```

- `RCPS_SingleLine` (in configuration: `SingleLine`) Try to put everything in the same line if possible. Otherwise normal line breaking rules take over.

```
// Fitting:
template <typename T> requires C<T> struct Foo {...

template <typename T> requires C<T> void bar(T t) {...

template <typename T> void bar(T t) requires C<T> {...

// Not fitting, one possible example:
template <typename LongName>
requires C<LongName>
struct Foo {...

template <typename LongName>
requires C<LongName>
void bar(LongName ln) {

template <typename LongName>
void bar(LongName ln)
    requires C<LongName> {
```

SeparateDefinitionBlocks (`SeparateDefinitionStyle`) clang-format 14

Specifies the use of empty lines to separate definition blocks, including classes, structs, enums, and functions.

```
Never                    v.s.     Always
#include <cstring>                 #include <cstring>
struct Foo {
  int a, b, c;                     struct Foo {
};                                   int a, b, c;
namespace Ns {                   };
class Bar {
public:                          namespace Ns {
  struct Foobar {                class Bar {
    int a;                       public:
    int b;                         struct Foobar {
  };                                 int a;
private:                             int b;
  int t;                         };
  int method1() {
    // ...                       private:
  }                                int t;
  enum List {
    ITEM1,                         int method1() {
    ITEM2                            // ...
  };                               }
  template<typename T>
  int method2(T x) {               enum List {
    // ...                           ITEM1,
  }                                  ITEM2
  int i, j, k;                     };
  int method3(int par) {
    // ...                         template<typename T>
  }                                int method2(T x) {
};                                   // ...
```

```
class C {};
}
```

```
  }

  int i, j, k;

  int method3(int par) {
    // ...
  }
};

class C {};
}
```

Possible values:

- `SDS_Leave` (in configuration: `Leave`) Leave definition blocks as they are.
- `SDS_Always` (in configuration: `Always`) Insert an empty line between definition blocks.
- `SDS_Never` (in configuration: `Never`) Remove any empty line between definition blocks.

ShortNamespaceLines (`Unsigned`) clang-format 13

The maximal number of unwrapped lines that a short namespace spans. Defaults to 1.

This determines the maximum length of short namespaces by counting unwrapped lines (i.e. containing neither opening nor closing namespace brace) and makes “FixNamespaceComments” omit adding end comments for those.

```
ShortNamespaceLines: 1     vs.     ShortNamespaceLines: 0
namespace a {                      namespace a {
  int foo;                           int foo;
}                                  } // namespace a

ShortNamespaceLines: 1     vs.     ShortNamespaceLines: 0
namespace b {                      namespace b {
  int foo;                           int foo;
  int bar;                           int bar;
} // namespace b                   } // namespace b
```

SortIncludes (`SortIncludesOptions`) clang-format 4

Controls if and how clang-format will sort `#includes`. If `Never`, includes are never sorted. If `CaseInsensitive`, includes are sorted in an ASCIIbetical or case insensitive fashion. If `CaseSensitive`, includes are sorted in an alphabetical or case sensitive fashion.

Possible values:

- `SI_Never` (in configuration: `Never`) Includes are never sorted.

```
#include "B/A.h"
#include "A/B.h"
#include "a/b.h"
#include "A/b.h"
#include "B/a.h"
```

- `SI_CaseSensitive` (in configuration: `CaseSensitive`) Includes are sorted in an ASCIIbetical or case sensitive fashion.

```
#include "A/B.h"
#include "A/b.h"
#include "B/A.h"
#include "B/a.h"
#include "a/b.h"
```

- `SI_CaseInsensitive` (in configuration: `CaseInsensitive`) Includes are sorted in an alphabetical or case insensitive fashion.

```
#include "A/B.h"
#include "A/b.h"
#include "a/b.h"
#include "B/A.h"
#include "B/a.h"
```

SortJavaStaticImport (`SortJavaStaticImportOptions`) clang-format 12

When sorting Java imports, by default static imports are placed before non-static imports. If `JavaStaticImportAfterImport` is `After`, static imports are placed after non-static imports.

Possible values:

- `SJSIO_Before` (in configuration: `Before`) Static imports are placed before non-static imports.

```
import static org.example.function1;

import org.example.ClassA;
```

- `SJSIO_After` (in configuration: `After`) Static imports are placed after non-static imports.

```
import org.example.ClassA;

import static org.example.function1;
```

SortUsingDeclarations (`Boolean`) clang-format 5

If `true`, clang-format will sort using declarations.

The order of using declarations is defined as follows: Split the strings by "::" and discard any initial empty strings. The last element of each list is a non-namespace name; all others are namespace names. Sort the lists of names lexicographically, where the sort order of individual names is that all non-namespace names come before all namespace names, and within those groups, names are in case-insensitive lexicographic order.

```
false:                                  true:
using std::cout;                vs.     using std::cin;
using std::cin;                         using std::cout;
```

SpaceAfterCStyleCast (`Boolean`) clang-format 3.5

If `true`, a space is inserted after C style casts.

```
true:                                   false:
(int) i;                        vs.     (int)i;
```

SpaceAfterLogicalNot (`Boolean`) clang-format 9

If `true`, a space is inserted after the logical not operator (`!`).

```
true:                                   false:
! someExpression();             vs.     !someExpression();
```

SpaceAfterTemplateKeyword (`Boolean`) clang-format 4

If `true`, a space will be inserted after the 'template' keyword.

```
true:                                   false:
template <int> void foo();      vs.     template<int> void foo();
```

SpaceAroundPointerQualifiers (`SpaceAroundPointerQualifiersStyle`) clang-format 12

Defines in which cases to put a space before or after pointer qualifiers

Possible values:

- `SAPQ_Default` (in configuration: `Default`) Don't ensure spaces around pointer qualifiers and use PointerAlignment instead.

```
PointerAlignment: Left                  PointerAlignment: Right
void* const* x = NULL;          vs.     void *const *x = NULL;
```

- `SAPQ_Before` (in configuration: `Before`) Ensure that there is a space before pointer qualifiers.

```
PointerAlignment: Left                  PointerAlignment: Right
void* const* x = NULL;          vs.     void * const *x = NULL;
```

- `SAPQ_After` (in configuration: `After`) Ensure that there is a space after pointer qualifiers.

```
PointerAlignment: Left                 PointerAlignment: Right
void* const * x = NULL;         vs.     void *const *x = NULL;
```

- `SAPQ_Both` (in configuration: `Both`) Ensure that there is a space both before and after pointer qualifiers.

```
PointerAlignment: Left                 PointerAlignment: Right
void* const * x = NULL;         vs.     void * const *x = NULL;
```

SpaceBeforeAssignmentOperators (`Boolean`) clang-format 3.7

If `false`, spaces will be removed before assignment operators.

```
true:                                  false:
int a = 5;                      vs.    int a= 5;
a += 42;                               a+= 42;
```

SpaceBeforeCaseColon (`Boolean`) clang-format 12

If `false`, spaces will be removed before case colon.

```
true:                                   false
switch (x) {                    vs.     switch (x) {
  case 1 : break;                         case 1: break;
}                                       }
```

SpaceBeforeCpp11BracedList (`Boolean`) clang-format 7

If `true`, a space will be inserted before a C++11 braced list used to initialize an object (after the preceding identifier or type).

```
true:                                  false:
Foo foo { bar };                vs.    Foo foo{ bar };
Foo {};                                Foo{};
vector<int> { 1, 2, 3 };               vector<int>{ 1, 2, 3 };
new int[3] { 1, 2, 3 };                new int[3]{ 1, 2, 3 };
```

SpaceBeforeCtorInitializerColon (`Boolean`) clang-format 7

If `false`, spaces will be removed before constructor initializer colon.

```
true:                                  false:
Foo::Foo() : a(a) {}                   Foo::Foo(): a(a) {}
```

SpaceBeforeInheritanceColon (`Boolean`) clang-format 7

If `false`, spaces will be removed before inheritance colon.

```
true:                                  false:
class Foo : Bar {}              vs.    class Foo: Bar {}
```

SpaceBeforeParens (`SpaceBeforeParensStyle`) clang-format 3.5

Defines in which cases to put a space before opening parentheses.

Possible values:

- `SBPO_Never` (in configuration: `Never`) Never put a space before opening parentheses.

```
void f() {
  if(true) {
    f();
  }
}
```

- `SBPO_ControlStatements` (in configuration: `ControlStatements`) Put a space before opening parentheses only after control statement keywords (`for/if/while...`).

```
void f() {
  if (true) {
    f();
  }
}
```

- `SBPO_ControlStatementsExceptControlMacros` (in configuration: `ControlStatementsExceptControlMacros`) Same as `SBPO_ControlStatements` except this option doesn't apply to ForEach and If macros. This is useful in projects where ForEach/If macros are treated as function calls instead of control statements. `SBPO_ControlStatementsExceptForEachMacros` remains an alias for backward compatibility.

```
void f() {
  Q_FOREACH(...) {
    f();
  }
}
```

- `SBPO_NonEmptyParentheses` (in configuration: `NonEmptyParentheses`) Put a space before opening parentheses only if the parentheses are not empty i.e. '()'

```
void() {
  if (true) {
    f();
    g (x, y, z);
  }
}
```

- `SBPO_Always` (in configuration: `Always`) Always put a space before opening parentheses, except when it's prohibited by the syntax rules (in function-like macro definitions) or when determined by other style rules (after unary operators, opening parentheses, etc.)

```
void f () {
  if (true) {
    f ();
  }
}
```

- `SBPO_Custom` (in configuration: `Custom`) Configure each individual space before parentheses in *SpaceBeforeParensOptions*.

SpaceBeforeParensOptions (`SpaceBeforeParensCustom`) clang-format 14

Control of individual space before parentheses.

If `SpaceBeforeParens` is set to `Custom`, use this to specify how each individual space before parentheses case should be handled. Otherwise, this is ignored.

```
# Example of usage:
SpaceBeforeParens: Custom
SpaceBeforeParensOptions:
  AfterControlStatements: true
  AfterFunctionDefinitionName: true
```

Nested configuration flags:

Precise control over the spacing before parentheses.

```
# Should be declared this way:
SpaceBeforeParens: Custom
SpaceBeforeParensOptions:
  AfterControlStatements: true
  AfterFunctionDefinitionName: true
```

- `bool AfterControlStatements` If `true`, put space betwee control statement keywords (for/if/while…) and opening parentheses.

```
true:                                   false:
if (...) {}                     vs.     if(...) {}
```

- `bool AfterForeachMacros` If `true`, put space between foreach macros and opening parentheses.

```
true:                                   false:
FOREACH (...)                   vs.     FOREACH(...)
  <loop-body>                             <loop-body>
```

- `bool AfterFunctionDeclarationName` If `true`, put a space between function declaration name and opening parentheses.

```
true:                                   false:
void f ();                      vs.     void f();
```

- `bool AfterFunctionDefinitionName` If `true`, put a space between function definition name and opening parentheses.

```
true:                                   false:
void f () {}                    vs.     void f() {}
```

- `bool AfterIfMacros` If `true`, put space between if macros and opening parentheses.

```
true:                                   false:
IF (...)                        vs.     IF(...)
  <conditional-body>                      <conditional-body>
```

- `bool AfterOverloadedOperator` If `true`, put a space between operator overloading and opening parentheses.

```
true:                                   false:
void operator++ (int a);        vs.     void operator++(int a);
object.operator++ (10);                 object.operator++(10);
```

- `bool AfterRequiresInClause` If `true`, put space between requires keyword in a requires clause and opening parentheses, if there is one.

```
true:                                   false:
template<typename T>            vs.     template<typename T>
requires (A<T> && B<T>)                 requires(A<T> && B<T>)
...                                     ...
```

- `bool AfterRequiresInExpression` If `true`, put space between requires keyword in a requires expression and opening parentheses.

```
true:                                   false:
template<typename T>            vs.     template<typename T>
concept C = requires (T t) {            concept C = requires(T t) {
              ...                                     ...
            }                                       }
```

- `bool BeforeNonEmptyParentheses` If `true`, put a space before opening parentheses only if the parentheses are not empty.

```
true:                                   false:
void f (int a);                 vs.     void f();
f (a);                                  f();
```

SpaceBeforeRangeBasedForLoopColon (`Boolean`) clang-format 7

If `false`, spaces will be removed before range-based for loop colon.

```
true:                                   false:
for (auto v : values) {}        vs.     for(auto v: values) {}
```

SpaceBeforeSquareBrackets (`Boolean`) clang-format 10

If `true`, spaces will be before `[`. Lambdas will not be affected. Only the first `[` will get a space added.

```
true:                                   false:
int a [5];                      vs.     int a[5];
int a [5][5];                   vs.     int a[5][5];
```

SpaceInEmptyBlock (`Boolean`) clang-format 10

If `true`, spaces will be inserted into `{}`.

```
true:                                   false:
void f() { }                    vs.     void f() {}
while (true) { }                        while (true) {}
```

SpaceInEmptyParentheses (`Boolean`) clang-format 3.7

If `true`, spaces may be inserted into `()`.

```
true:                                   false:
void f( ) {                     vs.     void f() {
  int x[] = {foo( ), bar( )};             int x[] = {foo(), bar()};
  if (true) {                             if (true) {
    f( );                                   f();
  }                                       }
}                                       }
```

SpacesBeforeTrailingComments (`Unsigned`) clang-format 3.7

The number of spaces before trailing line comments (`//` - comments).

This does not affect trailing block comments (`/*` - comments) as those commonly have different usage patterns and a number of special cases.

```
SpacesBeforeTrailingComments: 3
void f() {
  if (true) {   // foo1
    f();        // bar
  }             // foo
}
```

SpacesInAngles (`SpacesInAnglesStyle`) clang-format 3.4

The SpacesInAnglesStyle to use for template argument lists.

Possible values:

- `SIAS_Never` (in configuration: `Never`) Remove spaces after < and before >.

  ```
  static_cast<int>(arg);
  std::function<void(int)> fct;
  ```

- `SIAS_Always` (in configuration: `Always`) Add spaces after < and before >.

  ```
  static_cast< int >(arg);
  std::function< void(int) > fct;
  ```

- `SIAS_Leave` (in configuration: `Leave`) Keep a single space after < and before > if any spaces were present. Option `Standard: Cpp03` takes precedence.

SpacesInCStyleCastParentheses (`Boolean`) clang-format 3.7

If `true`, spaces may be inserted into C style casts.

```
true:                                   false:
x = ( int32 )y                  vs.     x = (int32)y
```

SpacesInConditionalStatement (`Boolean`) clang-format 10

If `true`, spaces will be inserted around if/for/switch/while conditions.

```
true:                                   false:
if ( a )  { ... }               vs.     if (a) { ... }
while ( i < 5 )  { ... }                while (i < 5) { ... }
```

SpacesInContainerLiterals (`Boolean`) clang-format 3.7

If `true`, spaces are inserted inside container literals (e.g. ObjC and Javascript array and dict literals).

```
true:                                   false:
var arr = [ 1, 2, 3 ];          vs.     var arr = [1, 2, 3];
f({a : 1, b : 2, c : 3});               f({a: 1, b: 2, c: 3});
```

SpacesInLineCommentPrefix (`SpacesInLineComment`) clang-format 13

How many spaces are allowed at the start of a line comment. To disable the maximum set it to `-1`, apart from that the maximum takes precedence over the minimum.

```
Minimum = 1
Maximum = -1
// One space is forced

//  but more spaces are possible

Minimum = 0
Maximum = 0
//Forces to start every comment directly after the slashes
```

Note that in line comment sections the relative indent of the subsequent lines is kept, that means the following:

```
before:                                   after:
Minimum: 1
//if (b) {                                // if (b) {
//  return true;                          //   return true;
//}                                       // }

Maximum: 0
/// List:                                 ///List:
///  - Foo                                /// - Foo
///    - Bar                              ///   - Bar
```

Nested configuration flags:

Control of spaces within a single line comment

- `unsigned Minimum` The minimum number of spaces at the start of the comment.
- `unsigned Maximum` The maximum number of spaces at the start of the comment.

SpacesInParentheses (`Boolean`) clang-format 3.7

If `true`, spaces will be inserted after `(` and before `)`.

```
true:                                   false:
t f( Deleted & ) & = delete;   vs.      t f(Deleted &) & = delete;
```

SpacesInSquareBrackets (`Boolean`) clang-format 3.7

If `true`, spaces will be inserted after `[` and before `]`. Lambdas without arguments or unspecified size array declarations will not be affected.

```
true:                                   false:
int a[ 5 ];                    vs.      int a[5];
std::unique_ptr<int[]> foo() {} // Won't be affected
```

Standard (`LanguageStandard`) clang-format 3.7

Parse and format C++ constructs compatible with this standard.

```
c++03:                                  latest:
vector<set<int> > x;           vs.      vector<set<int>> x;
```

Possible values:

- `LS_Cpp03` (in configuration: `c++03`) Parse and format as C++03. `Cpp03` is a deprecated alias for `c++03`
- `LS_Cpp11` (in configuration: `c++11`) Parse and format as C++11.
- `LS_Cpp14` (in configuration: `c++14`) Parse and format as C++14.
- `LS_Cpp17` (in configuration: `c++17`) Parse and format as C++17.
- `LS_Cpp20` (in configuration: `c++20`) Parse and format as C++20.
- `LS_Latest` (in configuration: `Latest`) Parse and format using the latest supported language version. `Cpp11` is a deprecated alias for `Latest`

- `LS_Auto` (in configuration: `Auto`) Automatic detection based on the input.

StatementAttributeLikeMacros (`List of Strings`) clang-format 12

Macros which are ignored in front of a statement, as if they were an attribute. So that they are not parsed as identifier, for example for Qts emit.

```
AlignConsecutiveDeclarations: true
StatementAttributeLikeMacros: []
unsigned char data = 'x';
emit          signal(data); // This is parsed as variable declaration.

AlignConsecutiveDeclarations: true
StatementAttributeLikeMacros: [emit]
unsigned char data = 'x';
emit signal(data); // Now it's fine again.
```

StatementMacros (`List of Strings`) clang-format 8

A vector of macros that should be interpreted as complete statements.

Typical macros are expressions, and require a semi-colon to be added; sometimes this is not the case, and this allows to make clang-format aware of such cases.

For example: Q_UNUSED

TabWidth (`Unsigned`) clang-format 3.7

The number of columns used for tab stops.

TypenameMacros (`List of Strings`) clang-format 9

A vector of macros that should be interpreted as type declarations instead of as function calls.

These are expected to be macros of the form:

```
STACK_OF(...)
```

In the .clang-format configuration file, this can be configured like:

```
TypenameMacros: ['STACK_OF', 'LIST']
```

For example: OpenSSL STACK_OF, BSD LIST_ENTRY.

UseCRLF (`Boolean`) clang-format 10

Use `\r\n` instead of `\n` for line breaks. Also used as fallback if `DeriveLineEnding` is true.

UseTab (`UseTabStyle`) clang-format 3.7

The way to use tab characters in the resulting file.

Possible values:

- `UT_Never` (in configuration: `Never`) Never use tab.
- `UT_ForIndentation` (in configuration: `ForIndentation`) Use tabs only for indentation.
- `UT_ForContinuationAndIndentation` (in configuration: `ForContinuationAndIndentation`) Fill all leading whitespace with tabs, and use spaces for alignment that appears within a line (e.g. consecutive assignments and declarations).
- `UT_AlignWithSpaces` (in configuration: `AlignWithSpaces`) Use tabs for line continuation and indentation, and spaces for alignment.
- `UT_Always` (in configuration: `Always`) Use tabs whenever we need to fill whitespace that spans at least from one tab stop to the next one.

WhitespaceSensitiveMacros (`List of Strings`) clang-format 11

A vector of macros which are whitespace-sensitive and should not be touched.

These are expected to be macros of the form:

```
STRINGIZE(...)
```

In the .clang-format configuration file, this can be configured like:

```
WhitespaceSensitiveMacros: ['STRINGIZE', 'PP_STRINGIZE']
```

For example: BOOST_PP_STRINGIZE

Adding additional style options

Each additional style option adds costs to the clang-format project. Some of these costs affect the clang-format development itself, as we need to make sure that any given combination of options work and that new features don't break any of the existing options in any way. There are also costs for end users as options become less discoverable and people have to think about and make a decision on options they don't really care about.

The goal of the clang-format project is more on the side of supporting a limited set of styles really well as opposed to supporting every single style used by a codebase somewhere in the wild. Of course, we do want to support all major projects and thus have established the following bar for adding style options. Each new style option must ..

- be used in a project of significant size (have dozens of contributors)
- have a publicly accessible style guide
- have a person willing to contribute and maintain patches

Examples

A style similar to the Linux Kernel style:

```
BasedOnStyle: LLVM
IndentWidth: 8
UseTab: Always
BreakBeforeBraces: Linux
AllowShortIfStatementsOnASingleLine: false
IndentCaseLabels: false
```

The result is (imagine that tabs are used for indentation here):

```
void test()
{
        switch (x) {
        case 0:
        case 1:
                do_something();
                break;
        case 2:
                do_something_else();
                break;
        default:
                break;
        }
        if (condition)
                do_something_completely_different();

        if (x == y) {
                q();
        } else if (x > y) {
                w();
        } else {
                r();
        }
}
```

A style similar to the default Visual Studio formatting style:

```
UseTab: Never
IndentWidth: 4
```

```
BreakBeforeBraces: Allman
AllowShortIfStatementsOnASingleLine: false
IndentCaseLabels: false
ColumnLimit: 0
```

The result is:

```
void test()
{
    switch (suffix)
    {
    case 0:
    case 1:
        do_something();
        break;
    case 2:
        do_something_else();
        break;
    default:
        break;
    }
    if (condition)
        do_something_completely_different();

    if (x == y)
    {
        q();
    }
    else if (x > y)
    {
        w();
    }
    else
    {
        r();
    }
}
```

Clang Formatted Status

Clang Formatted Status describes the state of LLVM source tree in terms of conformance to ClangFormat as of: March 06, 2022 17:32:26 (830ba4cebe79).

LLVM Clang-Format Status

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
bolt/include/bolt/Core	*15*	*10*	*5*	66%
bolt/include/bolt/Passes	*47*	*47*	*0*	100%
bolt/include/bolt/Profile	*8*	*8*	*0*	100%
bolt/include/bolt/Rewrite	*5*	*4*	*1*	80%
bolt/include/bolt/RuntimeLibs	*3*	*3*	*0*	100%
bolt/include/bolt/Utils	*4*	*4*	*0*	100%
bolt/lib/Core	*14*	*5*	*9*	35%
bolt/lib/Passes	*45*	*21*	*24*	46%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
bolt/lib/Profile	7	3	4	42%
bolt/lib/Rewrite	6	0	6	0%
bolt/lib/RuntimeLibs	3	3	0	100%
bolt/lib/Target/AArch64	1	0	1	0%
bolt/lib/Target/X86	1	0	1	0%
bolt/lib/Utils	2	1	1	50%
bolt/runtime	3	0	3	0%
bolt/tools/driver	1	0	1	0%
bolt/tools/heatmap	1	1	0	100%
bolt/tools/llvm-bolt-fuzzer	1	1	0	100%
bolt/tools/merge-fdata	1	0	1	0%
bolt/unittests/Core	1	1	0	100%
clang/bindings/python/tests/cindex/INPUTS	5	3	2	60%
clang/docs/analyzer/checkers	2	0	2	0%
clang/examples/AnnotateFunctions	1	0	1	0%
clang/examples/Attribute	1	1	0	100%
clang/examples/CallSuperAttribute	1	1	0	100%
clang/examples/PluginsOrder	1	1	0	100%
clang/examples/PrintFunctionNames	1	0	1	0%
clang/include/clang/Analysis	16	4	12	25%
clang/include/clang/Analysis/Analyses	15	3	12	20%
clang/include/clang/Analysis/DomainSpecific	2	0	2	0%
clang/include/clang/Analysis/FlowSensitive	16	15	1	93%
clang/include/clang/Analysis/Support	1	0	1	0%
clang/include/clang/APINotes	2	2	0	100%
clang/include/clang/ARCMigrate	3	0	3	0%
clang/include/clang/AST	114	20	94	17%
clang/include/clang/ASTMatchers	5	1	4	20%
clang/include/clang/ASTMatchers/Dynamic	4	1	3	25%
clang/include/clang/Basic	82	32	50	39%
clang/include/clang/CodeGen	9	0	9	0%
clang/include/clang/CrossTU	2	1	1	50%
clang/include/clang/DirectoryWatcher	1	1	0	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
clang/include/clang/Driver	17	4	13	23%
clang/include/clang/Edit	5	1	4	20%
clang/include/clang/Format	1	1	0	100%
clang/include/clang/Frontend	28	7	21	25%
clang/include/clang/FrontendTool	1	0	1	0%
clang/include/clang/Index	7	2	5	28%
clang/include/clang/IndexSerialization	1	1	0	100%
clang/include/clang/Interpreter	2	2	0	100%
clang/include/clang/Lex	29	6	23	20%
clang/include/clang/Parse	5	2	3	40%
clang/include/clang/Rewrite/Core	6	0	6	0%
clang/include/clang/Rewrite/Frontend	4	0	4	0%
clang/include/clang/Sema	32	3	29	9%
clang/include/clang/Serialization	14	3	11	21%
clang/include/clang/StaticAnalyzer/Checkers	4	1	3	25%
clang/include/clang/StaticAnalyzer/Core	5	1	4	20%
clang/include/clang/StaticAnalyzer/Core/BugReporter	4	1	3	25%
clang/include/clang/StaticAnalyzer/Core/PathSensitive	37	10	27	27%
clang/include/clang/StaticAnalyzer/Frontend	5	2	3	40%
clang/include/clang/Testing	2	2	0	100%
clang/include/clang/Tooling	17	10	7	58%
clang/include/clang/Tooling/ASTDiff	2	2	0	100%
clang/include/clang/Tooling/Core	2	0	2	0%
clang/include/clang/Tooling/DependencyScanning	5	5	0	100%
clang/include/clang/Tooling/Inclusions	3	3	0	100%
clang/include/clang/Tooling/Refactoring	15	12	3	80%
clang/include/clang/Tooling/Refactoring/Extract	2	2	0	100%
clang/include/clang/Tooling/Refactoring/Rename	6	5	1	83%
clang/include/clang/Tooling/Syntax	5	5	0	100%
clang/include/clang/Tooling/Syntax/Pseudo	5	5	0	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
clang/include/clang/Tooling/Transformer	8	6	2	75%
clang/include/clang-c	10	3	7	30%
clang/INPUTS	2	0	2	0%
clang/lib/Analysis	28	3	25	10%
clang/lib/Analysis/FlowSensitive	7	7	0	100%
clang/lib/Analysis/plugins/CheckerDependencyHandling	1	1	0	100%
clang/lib/Analysis/plugins/CheckerOptionHandling	1	0	1	0%
clang/lib/Analysis/plugins/SampleAnalyzer	1	1	0	100%
clang/lib/APINotes	3	3	0	100%
clang/lib/ARCMigrate	22	0	22	0%
clang/lib/AST	81	2	79	2%
clang/lib/AST/Interp	44	18	26	40%
clang/lib/ASTMatchers	3	1	2	33%
clang/lib/ASTMatchers/Dynamic	6	1	5	16%
clang/lib/Basic	39	13	26	33%
clang/lib/Basic/Targets	50	25	25	50%
clang/lib/CodeGen	87	9	78	10%
clang/lib/CrossTU	1	0	1	0%
clang/lib/DirectoryWatcher	2	2	0	100%
clang/lib/DirectoryWatcher/default	1	0	1	0%
clang/lib/DirectoryWatcher/linux	1	0	1	0%
clang/lib/DirectoryWatcher/mac	1	0	1	0%
clang/lib/DirectoryWatcher/windows	1	0	1	0%
clang/lib/Driver	14	2	12	14%
clang/lib/Driver/ToolChains	94	41	53	43%
clang/lib/Driver/ToolChains/Arch	20	7	13	35%
clang/lib/Edit	3	0	3	0%
clang/lib/Format	35	35	0	100%
clang/lib/Frontend	32	4	28	12%
clang/lib/Frontend/Rewrite	8	0	8	0%
clang/lib/FrontendTool	1	0	1	0%
clang/lib/Headers	146	14	132	9%
clang/lib/Headers/openmp_wrappers	5	4	1	80%
clang/lib/Headers/ppc_wrappers	7	2	5	28%
clang/lib/Index	11	2	9	18%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
clang/lib/IndexSerialization	*1*	*1*	*0*	100%
clang/lib/Interpreter	*5*	*5*	*0*	100%
clang/lib/Lex	*24*	*1*	*23*	4%
clang/lib/Parse	*15*	*1*	*14*	6%
clang/lib/Rewrite	*5*	*0*	*5*	0%
clang/lib/Sema	*55*	*4*	*51*	7%
clang/lib/Serialization	*17*	*2*	*15*	11%
clang/lib/StaticAnalyzer/Checkers	*122*	*19*	*103*	15%
clang/lib/StaticAnalyzer/Checkers/cert	*2*	*2*	*0*	100%
clang/lib/StaticAnalyzer/Checkers/MPI-Checker	*6*	*0*	*6*	0%
clang/lib/StaticAnalyzer/Checkers/RetainCountChecker	*4*	*0*	*4*	0%
clang/lib/StaticAnalyzer/Checkers/UninitializedObject	*3*	*1*	*2*	33%
clang/lib/StaticAnalyzer/Checkers/WebKit	*10*	*8*	*2*	80%
clang/lib/StaticAnalyzer/Core	*47*	*10*	*37*	21%
clang/lib/StaticAnalyzer/Frontend	*8*	*3*	*5*	37%
clang/lib/Testing	*1*	*1*	*0*	100%
clang/lib/Tooling	*16*	*7*	*9*	43%
clang/lib/Tooling/ASTDiff	*1*	*0*	*1*	0%
clang/lib/Tooling/Core	*2*	*0*	*2*	0%
clang/lib/Tooling/DependencyScanning	*5*	*4*	*1*	80%
clang/lib/Tooling/DumpTool	*4*	*3*	*1*	75%
clang/lib/Tooling/Inclusions	*3*	*3*	*0*	100%
clang/lib/Tooling/Refactoring	*5*	*3*	*2*	60%
clang/lib/Tooling/Refactoring/Extract	*2*	*1*	*1*	50%
clang/lib/Tooling/Refactoring/Rename	*5*	*2*	*3*	40%
clang/lib/Tooling/Syntax	*7*	*6*	*1*	85%
clang/lib/Tooling/Syntax/Pseudo	*8*	*8*	*0*	100%
clang/lib/Tooling/Transformer	*7*	*4*	*3*	57%
clang/tools/amdgpu-arch	*1*	*1*	*0*	100%
clang/tools/apinotes-test	*1*	*1*	*0*	100%
clang/tools/arcmt-test	*1*	*0*	*1*	0%
clang/tools/c-index-test	*1*	*0*	*1*	0%
clang/tools/clang-check	*1*	*0*	*1*	0%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
clang/tools/clang-diff	*1*	*0*	*1*	0%
clang/tools/clang-extdef-mapping	*1*	*0*	*1*	0%
clang/tools/clang-format	*1*	*1*	*0*	100%
clang/tools/clang-format/fuzzer	*1*	*0*	*1*	0%
clang/tools/clang-fuzzer	*6*	*4*	*2*	66%
clang/tools/clang-fuzzer/fuzzer-initialize	*2*	*0*	*2*	0%
clang/tools/clang-fuzzer/handle-cxx	*2*	*0*	*2*	0%
clang/tools/clang-fuzzer/handle-llvm	*3*	*1*	*2*	33%
clang/tools/clang-fuzzer/proto-to-cxx	*5*	*0*	*5*	0%
clang/tools/clang-fuzzer/proto-to-llvm	*3*	*0*	*3*	0%
clang/tools/clang-import-test	*1*	*0*	*1*	0%
clang/tools/clang-linker-wrapper	*3*	*2*	*1*	66%
clang/tools/clang-nvlink-wrapper	*1*	*1*	*0*	100%
clang/tools/clang-offload-bundler	*1*	*0*	*1*	0%
clang/tools/clang-offload-wrapper	*1*	*1*	*0*	100%
clang/tools/clang-pseudo	*1*	*1*	*0*	100%
clang/tools/clang-refactor	*4*	*4*	*0*	100%
clang/tools/clang-rename	*1*	*1*	*0*	100%
clang/tools/clang-repl	*1*	*1*	*0*	100%
clang/tools/clang-scan-deps	*1*	*1*	*0*	100%
clang/tools/clang-shlib	*1*	*1*	*0*	100%
clang/tools/diagtool	*9*	*0*	*9*	0%
clang/tools/driver	*4*	*1*	*3*	25%
clang/tools/libclang	*35*	*5*	*30*	14%
clang/tools/scan-build-py/tests/functional/src/include	*1*	*1*	*0*	100%
clang/unittests/Analysis	*6*	*2*	*4*	33%
clang/unittests/Analysis/FlowSensitive	*14*	*13*	*1*	92%
clang/unittests/AST	*30*	*8*	*22*	26%
clang/unittests/ASTMatchers	*6*	*3*	*3*	50%
clang/unittests/ASTMatchers/Dynamic	*3*	*0*	*3*	0%
clang/unittests/Basic	*8*	*4*	*4*	50%
clang/unittests/CodeGen	*6*	*1*	*5*	16%
clang/unittests/CrossTU	*1*	*1*	*0*	100%
clang/unittests/DirectoryWatcher	*1*	*0*	*1*	0%
clang/unittests/Driver	*5*	*1*	*4*	20%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
clang/unittests/Format	*24*	*24*	*0*	100%
clang/unittests/Frontend	*11*	*7*	*4*	63%
clang/unittests/Index	*1*	*1*	*0*	100%
clang/unittests/Interpreter	*2*	*2*	*0*	100%
clang/unittests/Interpreter/ExceptionTests	*1*	*0*	*1*	0%
clang/unittests/Introspection	*1*	*0*	*1*	0%
clang/unittests/Lex	*8*	*4*	*4*	50%
clang/unittests/libclang	*2*	*0*	*2*	0%
clang/unittests/libclang/CrashTests	*1*	*1*	*0*	100%
clang/unittests/Rename	*6*	*0*	*6*	0%
clang/unittests/Rewrite	*2*	*1*	*1*	50%
clang/unittests/Sema	*3*	*2*	*1*	66%
clang/unittests/Serialization	*2*	*2*	*0*	100%
clang/unittests/StaticAnalyzer	*16*	*7*	*9*	43%
clang/unittests/Tooling	*30*	*10*	*20*	33%
clang/unittests/Tooling/RecursiveASTVisitorTests	*30*	*12*	*18*	40%
clang/unittests/Tooling/Syntax	*7*	*3*	*4*	42%
clang/unittests/Tooling/Syntax/Pseudo	*4*	*4*	*0*	100%
clang/utils/perf-training/cxx	*1*	*0*	*1*	0%
clang/utils/TableGen	*22*	*3*	*19*	13%
clang-tools-extra/clang-apply-replacements/include/clang-apply-replacements/Tooling	*1*	*1*	*0*	100%
clang-tools-extra/clang-apply-replacements/lib/Tooling	*1*	*1*	*0*	100%
clang-tools-extra/clang-apply-replacements/tool	*1*	*1*	*0*	100%
clang-tools-extra/clang-change-namespace	*2*	*0*	*2*	0%
clang-tools-extra/clang-change-namespace/tool	*1*	*0*	*1*	0%
clang-tools-extra/clang-doc	*17*	*16*	*1*	94%
clang-tools-extra/clang-doc/tool	*1*	*1*	*0*	100%
clang-tools-extra/clang-include-fixer	*13*	*8*	*5*	61%
clang-tools-extra/clang-include-fixer/find-all-symbols	*17*	*13*	*4*	76%
clang-tools-extra/clang-include-fixer/find-all-symbols/tool	*1*	*0*	*1*	0%
clang-tools-extra/clang-include-fixer/plugin	*1*	*1*	*0*	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
clang-tools-extra/clang-include-fixer/tool	*1*	*0*	*1*	0%
clang-tools-extra/clang-move	*4*	*1*	*3*	25%
clang-tools-extra/clang-move/tool	*1*	*1*	*0*	100%
clang-tools-extra/clang-query	*5*	*4*	*1*	80%
clang-tools-extra/clang-query/tool	*1*	*0*	*1*	0%
clang-tools-extra/clang-reorder-fields	*2*	*1*	*1*	50%
clang-tools-extra/clang-reorder-fields/tool	*1*	*0*	*1*	0%
clang-tools-extra/clang-tidy	*20*	*14*	*6*	70%
clang-tools-extra/clang-tidy/abseil	*42*	*31*	*11*	73%
clang-tools-extra/clang-tidy/altera	*11*	*9*	*2*	81%
clang-tools-extra/clang-tidy/android	*33*	*23*	*10*	69%
clang-tools-extra/clang-tidy/boost	*3*	*3*	*0*	100%
clang-tools-extra/clang-tidy/bugprone	*125*	*106*	*19*	84%
clang-tools-extra/clang-tidy/cert	*29*	*28*	*1*	96%
clang-tools-extra/clang-tidy/concurrency	*5*	*4*	*1*	80%
clang-tools-extra/clang-tidy/cppcoreguidelines	*45*	*42*	*3*	93%
clang-tools-extra/clang-tidy/darwin	*5*	*2*	*3*	40%
clang-tools-extra/clang-tidy/fuchsia	*15*	*10*	*5*	66%
clang-tools-extra/clang-tidy/google	*33*	*22*	*11*	66%
clang-tools-extra/clang-tidy/hicpp	*9*	*7*	*2*	77%
clang-tools-extra/clang-tidy/linuxkernel	*3*	*2*	*1*	66%
clang-tools-extra/clang-tidy/llvm	*11*	*10*	*1*	90%
clang-tools-extra/clang-tidy/llvmlibc	*7*	*7*	*0*	100%
clang-tools-extra/clang-tidy/misc	*33*	*30*	*3*	90%
clang-tools-extra/clang-tidy/modernize	*67*	*48*	*19*	71%
clang-tools-extra/clang-tidy/mpi	*5*	*5*	*0*	100%
clang-tools-extra/clang-tidy/objc	*17*	*12*	*5*	70%
clang-tools-extra/clang-tidy/openmp	*5*	*5*	*0*	100%
clang-tools-extra/clang-tidy/performance	*31*	*24*	*7*	77%
clang-tools-extra/clang-tidy/plugin	*1*	*1*	*0*	100%
clang-tools-extra/clang-tidy/portability	*5*	*3*	*2*	60%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
clang-tools-extra/clang-tidy/readability	*88*	*76*	*12*	86%
clang-tools-extra/clang-tidy/tool	*3*	*2*	*1*	66%
clang-tools-extra/clang-tidy/utils	*35*	*31*	*4*	88%
clang-tools-extra/clang-tidy/zircon	*3*	*3*	*0*	100%
clang-tools-extra/clangd	*97*	*81*	*16*	83%
clang-tools-extra/clangd/benchmarks	*1*	*1*	*0*	100%
clang-tools-extra/clangd/benchmarks/CompletionModel	*1*	*0*	*1*	0%
clang-tools-extra/clangd/fuzzer	*2*	*2*	*0*	100%
clang-tools-extra/clangd/index	*39*	*36*	*3*	92%
clang-tools-extra/clangd/index/dex	*9*	*7*	*2*	77%
clang-tools-extra/clangd/index/dex/dexp	*1*	*1*	*0*	100%
clang-tools-extra/clangd/index/remote	*2*	*2*	*0*	100%
clang-tools-extra/clangd/index/remote/marshalling	*2*	*2*	*0*	100%
clang-tools-extra/clangd/index/remote/monitor	*1*	*1*	*0*	100%
clang-tools-extra/clangd/index/remote/server	*1*	*1*	*0*	100%
clang-tools-extra/clangd/index/remote/unimplemented	*1*	*1*	*0*	100%
clang-tools-extra/clangd/indexer	*1*	*1*	*0*	100%
clang-tools-extra/clangd/refactor	*6*	*5*	*1*	83%
clang-tools-extra/clangd/refactor/tweaks	*14*	*10*	*4*	71%
clang-tools-extra/clangd/support	*25*	*24*	*1*	96%
clang-tools-extra/clangd/tool	*2*	*2*	*0*	100%
clang-tools-extra/clangd/unittests	*79*	*66*	*13*	83%
clang-tools-extra/clangd/unittests/decision_forest_model	*1*	*1*	*0*	100%
clang-tools-extra/clangd/unittests/remote	*1*	*1*	*0*	100%
clang-tools-extra/clangd/unittests/support	*11*	*11*	*0*	100%
clang-tools-extra/clangd/unittests/tweaks	*20*	*19*	*1*	95%
clang-tools-extra/clangd/unittests/xpc	*1*	*1*	*0*	100%
clang-tools-extra/clangd/xpc	*3*	*3*	*0*	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
clang-tools-extra/clangd/xpc/framework	*1*	*1*	*0*	100%
clang-tools-extra/clangd/xpc/test-client	*1*	*1*	*0*	100%
clang-tools-extra/modularize	*9*	*1*	*8*	11%
clang-tools-extra/pp-trace	*3*	*1*	*2*	33%
clang-tools-extra/tool-template	*1*	*1*	*0*	100%
clang-tools-extra/unittests/clang-apply-replacements	*1*	*1*	*0*	100%
clang-tools-extra/unittests/clang-change-namespace	*1*	*0*	*1*	0%
clang-tools-extra/unittests/clang-doc	*9*	*9*	*0*	100%
clang-tools-extra/unittests/clang-include-fixer	*2*	*0*	*2*	0%
clang-tools-extra/unittests/clang-include-fixer/find-all-symbols	*1*	*0*	*1*	0%
clang-tools-extra/unittests/clang-move	*1*	*0*	*1*	0%
clang-tools-extra/unittests/clang-query	*2*	*0*	*2*	0%
clang-tools-extra/unittests/clang-tidy	*16*	*9*	*7*	56%
clang-tools-extra/unittests/include/common	*1*	*0*	*1*	0%
compiler-rt/include/fuzzer	*1*	*0*	*1*	0%
compiler-rt/include/sanitizer	*15*	*3*	*12*	20%
compiler-rt/include/xray	*3*	*2*	*1*	66%
compiler-rt/lib/asan	*57*	*5*	*52*	8%
compiler-rt/lib/asan/tests	*17*	*1*	*16*	5%
compiler-rt/lib/BlocksRuntime	*2*	*0*	*2*	0%
compiler-rt/lib/builtins	*11*	*9*	*2*	81%
compiler-rt/lib/builtins/arm	*1*	*0*	*1*	0%
compiler-rt/lib/builtins/ppc	*1*	*1*	*0*	100%
compiler-rt/lib/cfi	*1*	*0*	*1*	0%
compiler-rt/lib/dfsan	*14*	*9*	*5*	64%
compiler-rt/lib/fuzzer	*47*	*9*	*38*	19%
compiler-rt/lib/fuzzer/afl	*1*	*0*	*1*	0%
compiler-rt/lib/fuzzer/dataflow	*3*	*0*	*3*	0%
compiler-rt/lib/fuzzer/tests	*2*	*1*	*1*	50%
compiler-rt/lib/gwp_asan	*12*	*12*	*0*	100%
compiler-rt/lib/gwp_asan/optional	*10*	*10*	*0*	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
compiler-rt/lib/gwp_asan/platform_specific	*13*	*13*	*0*	100%
compiler-rt/lib/gwp_asan/tests	*15*	*14*	*1*	93%
compiler-rt/lib/gwp_asan/tests/platform_specific	*1*	*1*	*0*	100%
compiler-rt/lib/hwasan	*30*	*9*	*21*	30%
compiler-rt/lib/interception	*8*	*1*	*7*	12%
compiler-rt/lib/interception/tests	*3*	*1*	*2*	33%
compiler-rt/lib/lsan	*20*	*4*	*16*	20%
compiler-rt/lib/memprof	*31*	*29*	*2*	93%
compiler-rt/lib/memprof/tests	*2*	*2*	*0*	100%
compiler-rt/lib/msan	*18*	*4*	*14*	22%
compiler-rt/lib/msan/tests	*4*	*0*	*4*	0%
compiler-rt/lib/orc	*21*	*16*	*5*	76%
compiler-rt/lib/orc/unittests	*10*	*9*	*1*	90%
compiler-rt/lib/profile	*6*	*0*	*6*	0%
compiler-rt/lib/safestack	*3*	*1*	*2*	33%
compiler-rt/lib/sanitizer_common	*167*	*29*	*138*	17%
compiler-rt/lib/sanitizer_common/symbolizer	*2*	*2*	*0*	100%
compiler-rt/lib/sanitizer_common/tests	*46*	*12*	*34*	26%
compiler-rt/lib/scudo	*20*	*0*	*20*	0%
compiler-rt/lib/scudo/standalone	*49*	*48*	*1*	97%
compiler-rt/lib/scudo/standalone/benchmarks	*1*	*1*	*0*	100%
compiler-rt/lib/scudo/standalone/fuzz	*1*	*1*	*0*	100%
compiler-rt/lib/scudo/standalone/include/scudo	*1*	*1*	*0*	100%
compiler-rt/lib/scudo/standalone/tests	*25*	*24*	*1*	96%
compiler-rt/lib/scudo/standalone/tools	*1*	*1*	*0*	100%
compiler-rt/lib/stats	*3*	*0*	*3*	0%
compiler-rt/lib/tsan/benchmarks	*6*	*0*	*6*	0%
compiler-rt/lib/tsan/dd	*3*	*0*	*3*	0%
compiler-rt/lib/tsan/go	*1*	*0*	*1*	0%
compiler-rt/lib/tsan/rtl	*59*	*14*	*45*	23%
compiler-rt/lib/tsan/rtl-old	*61*	*13*	*48*	21%
compiler-rt/lib/tsan/tests/rtl	*10*	*0*	*10*	0%
compiler-rt/lib/tsan/tests/unit	*11*	*3*	*8*	27%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
compiler-rt/lib/ubsan	*27*	*7*	*20*	25%
compiler-rt/lib/ubsan_minimal	*1*	*0*	*1*	0%
compiler-rt/lib/xray	*40*	*27*	*13*	67%
compiler-rt/lib/xray/tests/unit	*10*	*8*	*2*	80%
compiler-rt/tools/gwp_asan	*2*	*2*	*0*	100%
cross-project-tests/debuginfo-tests/clang_llvm_roundtrip	*2*	*1*	*1*	50%
cross-project-tests/debuginfo-tests/dexter/feature_tests/commands/penalty	*10*	*0*	*10*	0%
cross-project-tests/debuginfo-tests/dexter/feature_tests/commands/perfect	*7*	*0*	*7*	0%
cross-project-tests/debuginfo-tests/dexter/feature_tests/commands/perfect/dex_declare_address	*7*	*0*	*7*	0%
cross-project-tests/debuginfo-tests/dexter/feature_tests/commands/perfect/dex_declare_file/dex_and_source	*1*	*1*	*0*	100%
cross-project-tests/debuginfo-tests/dexter/feature_tests/commands/perfect/dex_declare_file/precompiled_binary	*1*	*1*	*0*	100%
cross-project-tests/debuginfo-tests/dexter/feature_tests/commands/perfect/dex_declare_file/precompiled_binary_different_dir/source	*1*	*1*	*0*	100%
cross-project-tests/debuginfo-tests/dexter/feature_tests/commands/perfect/dex_declare_file/windows_non canonical_path/source	*1*	*0*	*1*	0%
cross-project-tests/debuginfo-tests/dexter/feature_tests/commands/perfect/dex_finish_test	*8*	*0*	*8*	0%
cross-project-tests/debuginfo-tests/dexter/feature_tests/commands/perfect/expect_step_kind	*5*	*0*	*5*	0%
cross-project-tests/debuginfo-tests/dexter/feature_tests/commands/perfect/limit_steps	*8*	*2*	*6*	25%
cross-project-tests/debuginfo-tests/dexter/feature_tests/subtools	*1*	*0*	*1*	0%
cross-project-tests/debuginfo-tests/dexter/feature_tests/subtools/clang-opt-bisect	*2*	*0*	*2*	0%
cross-project-tests/debuginfo-tests/dexter-tests	*15*	*3*	*12*	20%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
cross-project-tests/debuginfo-tests/lldb-tests	8	0	8	0%
cross-project-tests/debuginfo-tests/llvm-prettyprinters/gdb	2	1	1	50%
flang/examples	1	1	0	100%
flang/examples/FlangOmpReport	3	3	0	100%
flang/examples/PrintFlangFunctionNames	1	1	0	100%
flang/include/flang	1	1	0	100%
flang/include/flang/Common	21	21	0	100%
flang/include/flang/Decimal	2	2	0	100%
flang/include/flang/Evaluate	23	23	0	100%
flang/include/flang/Frontend	11	10	1	90%
flang/include/flang/FrontendTool	1	1	0	100%
flang/include/flang/Lower	25	24	1	96%
flang/include/flang/Lower/Support	2	2	0	100%
flang/include/flang/Optimizer/Builder	7	7	0	100%
flang/include/flang/Optimizer/Builder/Runtime	10	10	0	100%
flang/include/flang/Optimizer/CodeGen	1	1	0	100%
flang/include/flang/Optimizer/Dialect	5	5	0	100%
flang/include/flang/Optimizer/Support	8	8	0	100%
flang/include/flang/Optimizer/Transforms	1	1	0	100%
flang/include/flang/Parser	17	16	1	94%
flang/include/flang/Runtime	28	27	1	96%
flang/include/flang/Semantics	9	8	1	88%
flang/lib/Common	4	4	0	100%
flang/lib/Decimal	3	3	0	100%
flang/lib/Evaluate	33	31	2	93%
flang/lib/Frontend	8	6	2	75%
flang/lib/FrontendTool	1	1	0	100%
flang/lib/Lower	20	20	0	100%
flang/lib/Optimizer/Builder	6	6	0	100%
flang/lib/Optimizer/Builder/Runtime	9	9	0	100%
flang/lib/Optimizer/CodeGen	10	10	0	100%
flang/lib/Optimizer/Dialect	5	5	0	100%
flang/lib/Optimizer/Support	4	4	0	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
flang/lib/Optimizer/Transforms	*10*	*10*	*0*	100%
flang/lib/Parser	*35*	*35*	*0*	100%
flang/lib/Semantics	*78*	*69*	*9*	88%
flang/module	*1*	*1*	*0*	100%
flang/runtime	*74*	*72*	*2*	97%
flang/tools/bbc	*1*	*1*	*0*	100%
flang/tools/f18	*1*	*1*	*0*	100%
flang/tools/f18-parse-demo	*2*	*2*	*0*	100%
flang/tools/fir-opt	*1*	*1*	*0*	100%
flang/tools/flang-driver	*2*	*2*	*0*	100%
flang/tools/tco	*1*	*1*	*0*	100%
flang/unittests/Common	*1*	*1*	*0*	100%
flang/unittests/Decimal	*2*	*2*	*0*	100%
flang/unittests/Evaluate	*15*	*15*	*0*	100%
flang/unittests/Frontend	*2*	*2*	*0*	100%
flang/unittests/Optimizer	*4*	*3*	*1*	75%
flang/unittests/Optimizer/Builder	*4*	*4*	*0*	100%
flang/unittests/Optimizer/Builder/Runtime	*10*	*10*	*0*	100%
flang/unittests/Runtime	*22*	*22*	*0*	100%
libc/AOR_v20.02/math	*4*	*1*	*3*	25%
libc/AOR_v20.02/math/include	*1*	*0*	*1*	0%
libc/AOR_v20.02/networking	*1*	*0*	*1*	0%
libc/AOR_v20.02/networking/include	*1*	*0*	*1*	0%
libc/AOR_v20.02/string	*1*	*0*	*1*	0%
libc/AOR_v20.02/string/include	*1*	*0*	*1*	0%
libc/benchmarks	*15*	*14*	*1*	93%
libc/benchmarks/automemcpy/include/automemcpy	*4*	*4*	*0*	100%
libc/benchmarks/automemcpy/lib	*5*	*5*	*0*	100%
libc/benchmarks/automemcpy/unittests	*2*	*2*	*0*	100%
libc/config/linux	*1*	*1*	*0*	100%
libc/fuzzing/math	*6*	*6*	*0*	100%
libc/fuzzing/stdlib	*3*	*3*	*0*	100%
libc/fuzzing/string	*3*	*2*	*1*	66%
libc/include	*1*	*1*	*0*	100%
libc/include/llvm-libc-macros	*2*	*2*	*0*	100%
libc/include/llvm-libc-macros/linux	*1*	*1*	*0*	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
libc/include/llvm-libc-types	28	28	0	100%
libc/loader/linux/aarch64	1	1	0	100%
libc/loader/linux/x86_64	1	1	0	100%
libc/src/assert	3	1	2	33%
libc/src/ctype	32	32	0	100%
libc/src/errno	4	4	0	100%
libc/src/fcntl	3	3	0	100%
libc/src/fcntl/linux	3	3	0	100%
libc/src/fenv	28	28	0	100%
libc/src/inttypes	6	6	0	100%
libc/src/math	91	91	0	100%
libc/src/math/aarch64	10	10	0	100%
libc/src/math/generic	94	94	0	100%
libc/src/math/x86_64	3	3	0	100%
libc/src/signal	8	8	0	100%
libc/src/signal/linux	10	10	0	100%
libc/src/stdio	3	3	0	100%
libc/src/stdlib	46	46	0	100%
libc/src/stdlib/linux	2	2	0	100%
libc/src/string	61	61	0	100%
libc/src/string/memory_utils	8	7	1	87%
libc/src/sys/mman	2	2	0	100%
libc/src/sys/mman/linux	2	1	1	50%
libc/src/sys/stat	2	2	0	100%
libc/src/sys/stat/linux	2	2	0	100%
libc/src/threads	16	16	0	100%
libc/src/threads/linux	11	7	4	63%
libc/src/time	12	12	0	100%
libc/src/unistd	7	7	0	100%
libc/src/unistd/linux	7	7	0	100%
libc/src/__support	10	10	0	100%
libc/src/__support/CPP	11	10	1	90%
libc/src/__support/File	2	2	0	100%
libc/src/__support/FPUtil	15	14	1	93%
libc/src/__support/FPUtil/aarch64	3	3	0	100%
libc/src/__support/FPUtil/generic	3	3	0	100%
libc/src/__support/FPUtil/x86_64	6	5	1	83%
libc/src/__support/OSUtil	3	3	0	100%
libc/src/__support/OSUtil/linux	3	2	1	66%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
libc/src/__support/OSUtil/linux/aarch64	*1*	*1*	*0*	100%
libc/src/__support/OSUtil/linux/x86_64	*1*	*1*	*0*	100%
libc/src/__support/threads	*1*	*1*	*0*	100%
libc/src/__support/threads/linux	*1*	*1*	*0*	100%
libc/utils/HdrGen	*9*	*9*	*0*	100%
libc/utils/HdrGen/PrototypeTestGen	*1*	*1*	*0*	100%
libc/utils/LibcTableGenUtil	*2*	*2*	*0*	100%
libc/utils/MPFRWrapper	*3*	*3*	*0*	100%
libc/utils/testutils	*10*	*9*	*1*	90%
libc/utils/tools/WrapperGen	*1*	*1*	*0*	100%
libc/utils/UnitTest	*12*	*11*	*1*	91%
libclc/generic/include	*2*	*1*	*1*	50%
libclc/generic/include/clc	*6*	*2*	*4*	33%
libclc/generic/include/clc/async	*4*	*4*	*0*	100%
libclc/generic/include/clc/atomic	*11*	*7*	*4*	63%
libclc/generic/include/clc/cl_khr_global_int32_base_atomics	*6*	*5*	*1*	83%
libclc/generic/include/clc/cl_khr_global_int32_extended_atomics	*5*	*5*	*0*	100%
libclc/generic/include/clc/cl_khr_int64_base_atomics	*6*	*3*	*3*	50%
libclc/generic/include/clc/cl_khr_int64_extended_atomics	*5*	*5*	*0*	100%
libclc/generic/include/clc/cl_khr_local_int32_base_atomics	*6*	*5*	*1*	83%
libclc/generic/include/clc/cl_khr_local_int32_extended_atomics	*5*	*5*	*0*	100%
libclc/generic/include/clc/common	*6*	*6*	*0*	100%
libclc/generic/include/clc/explicit_fence	*1*	*1*	*0*	100%
libclc/generic/include/clc/float	*1*	*0*	*1*	0%
libclc/generic/include/clc/geometric	*8*	*8*	*0*	100%
libclc/generic/include/clc/image	*2*	*0*	*2*	0%
libclc/generic/include/clc/integer	*16*	*13*	*3*	81%
libclc/generic/include/clc/math	*95*	*92*	*3*	96%
libclc/generic/include/clc/misc	*2*	*0*	*2*	0%
libclc/generic/include/clc/relational	*18*	*12*	*6*	66%
libclc/generic/include/clc/shared	*5*	*3*	*2*	60%
libclc/generic/include/clc/synchronization	*2*	*2*	*0*	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
libclc/generic/include/clc/workitem	8	8	0	100%
libclc/generic/include/integer	1	1	0	100%
libclc/generic/include/math	15	15	0	100%
libclc/generic/lib	1	0	1	0%
libclc/generic/lib/math	8	1	7	12%
libclc/generic/lib/relational	1	0	1	0%
libclc/utils	1	0	1	0%
libcxx/benchmarks	28	10	18	35%
libcxx/include	22	0	22	0%
libcxx/include/__algorithm	102	15	87	14%
libcxx/include/__bit	2	0	2	0%
libcxx/include/__charconv	3	0	3	0%
libcxx/include/__chrono	8	0	8	0%
libcxx/include/__compare	13	1	12	7%
libcxx/include/__concepts	22	0	22	0%
libcxx/include/__coroutine	4	0	4	0%
libcxx/include/__filesystem	16	3	13	18%
libcxx/include/__format	17	2	15	11%
libcxx/include/__functional	27	0	27	0%
libcxx/include/__ios	1	0	1	0%
libcxx/include/__iterator	36	0	36	0%
libcxx/include/__memory	19	1	18	5%
libcxx/include/__numeric	13	4	9	30%
libcxx/include/__random	37	2	35	5%
libcxx/include/__ranges	29	2	27	6%
libcxx/include/__support/android	1	0	1	0%
libcxx/include/__support/fuchsia	1	0	1	0%
libcxx/include/__support/ibm	6	2	4	33%
libcxx/include/__support/musl	1	0	1	0%
libcxx/include/__support/newlib	1	0	1	0%
libcxx/include/__support/openbsd	1	1	0	100%
libcxx/include/__support/solaris	3	2	1	66%
libcxx/include/__support/win32	2	0	2	0%
libcxx/include/__support/xlocale	3	0	3	0%
libcxx/include/__thread	2	0	2	0%
libcxx/include/__utility	17	5	12	29%
libcxx/include/__variant	1	0	1	0%
libcxx/src	42	6	36	14%
libcxx/src/experimental	2	1	1	50%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
libcxx/src/filesystem	5	0	5	0%
libcxx/src/include	6	1	5	16%
libcxx/src/include/ryu	9	8	1	88%
libcxx/src/ryu	3	3	0	100%
libcxx/src/support/ibm	3	0	3	0%
libcxx/src/support/solaris	1	0	1	0%
libcxx/src/support/win32	3	0	3	0%
libcxxabi/fuzz	1	0	1	0%
libcxxabi/include	2	0	2	0%
libcxxabi/src	25	1	24	4%
libcxxabi/src/demangle	4	2	2	50%
libunwind/include	5	0	5	0%
libunwind/include/mach-o	1	0	1	0%
libunwind/src	10	1	9	10%
lld/COFF	37	13	24	35%
lld/Common	11	9	2	81%
lld/ELF	48	25	23	52%
lld/ELF/Arch	14	4	10	28%
lld/include/lld/Common	14	8	6	57%
lld/include/lld/Core	20	4	16	20%
lld/MachO	45	43	2	95%
lld/MachO/Arch	6	6	0	100%
lld/MinGW	1	1	0	100%
lld/tools/lld	1	1	0	100%
lld/wasm	29	15	14	51%
lldb/bindings/python	1	1	0	100%
lldb/examples/darwin/heap_find/heap	1	1	0	100%
lldb/examples/functions	1	0	1	0%
lldb/examples/interposing/darwin/fd_interposing	1	0	1	0%
lldb/examples/lookup	1	0	1	0%
lldb/examples/plugins/commands	1	1	0	100%
lldb/examples/synthetic/bitfield	1	1	0	100%
lldb/include/lldb	12	6	6	50%
lldb/include/lldb/API	70	60	10	85%
lldb/include/lldb/Breakpoint	25	9	16	36%
lldb/include/lldb/Core	61	31	30	50%
lldb/include/lldb/DataFormatters	18	10	8	55%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
lldb/include/lldb/Expression	17	7	10	41%
lldb/include/lldb/Host	39	20	19	51%
lldb/include/lldb/Host/android	1	1	0	100%
lldb/include/lldb/Host/common	8	2	6	25%
lldb/include/lldb/Host/freebsd	1	0	1	0%
lldb/include/lldb/Host/linux	6	4	2	66%
lldb/include/lldb/Host/macosx	2	0	2	0%
lldb/include/lldb/Host/netbsd	1	0	1	0%
lldb/include/lldb/Host/openbsd	1	0	1	0%
lldb/include/lldb/Host/posix	9	7	2	77%
lldb/include/lldb/Host/windows	10	4	6	40%
lldb/include/lldb/Initialization	3	1	2	33%
lldb/include/lldb/Interpreter	49	36	13	73%
lldb/include/lldb/Symbol	35	14	21	40%
lldb/include/lldb/Target	78	51	27	65%
lldb/include/lldb/Utility	63	41	22	65%
lldb/include/lldb/Version	1	1	0	100%
lldb/source/API	73	36	37	49%
lldb/source/Breakpoint	24	6	18	25%
lldb/source/Commands	70	57	13	81%
lldb/source/Core	49	26	23	53%
lldb/source/DataFormatters	16	3	13	18%
lldb/source/Expression	13	5	8	38%
lldb/source/Host/android	2	2	0	100%
lldb/source/Host/common	31	16	15	51%
lldb/source/Host/freebsd	2	2	0	100%
lldb/source/Host/linux	5	5	0	100%
lldb/source/Host/macosx/cfcpp	14	12	2	85%
lldb/source/Host/macosx/objcxx	1	1	0	100%
lldb/source/Host/netbsd	2	0	2	0%
lldb/source/Host/openbsd	2	1	1	50%
lldb/source/Host/posix	9	6	3	66%
lldb/source/Host/windows	11	7	4	63%
lldb/source/Initialization	3	3	0	100%
lldb/source/Interpreter	44	24	20	54%
lldb/source/Plugins/ABI/AArch64	6	3	3	50%
lldb/source/Plugins/ABI/ARC	2	0	2	0%
lldb/source/Plugins/ABI/ARM	6	2	4	33%
lldb/source/Plugins/ABI/Hexagon	2	0	2	0%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
lldb/source/Plugins/ABI/Mips	*6*	*2*	*4*	33%
lldb/source/Plugins/ABI/PowerPC	*6*	*3*	*3*	50%
lldb/source/Plugins/ABI/SystemZ	*2*	*0*	*2*	0%
lldb/source/Plugins/ABI/X86	*13*	*4*	*9*	30%
lldb/source/Plugins/Architecture/AArch64	*2*	*2*	*0*	100%
lldb/source/Plugins/Architecture/Arm	*2*	*1*	*1*	50%
lldb/source/Plugins/Architecture/Mips	*2*	*0*	*2*	0%
lldb/source/Plugins/Architecture/PPC64	*2*	*2*	*0*	100%
lldb/source/Plugins/Disassembler/LLVMC	*2*	*1*	*1*	50%
lldb/source/Plugins/DynamicLoader/Darwin-Kernel	*2*	*0*	*2*	0%
lldb/source/Plugins/DynamicLoader/Hexagon-DYLD	*4*	*3*	*1*	75%
lldb/source/Plugins/DynamicLoader/MacOSX-DYLD	*6*	*3*	*3*	50%
lldb/source/Plugins/DynamicLoader/POSIX-DYLD	*4*	*2*	*2*	50%
lldb/source/Plugins/DynamicLoader/Static	*2*	*1*	*1*	50%
lldb/source/Plugins/DynamicLoader/wasm-DYLD	*2*	*2*	*0*	100%
lldb/source/Plugins/DynamicLoader/Windows-DYLD	*2*	*1*	*1*	50%
lldb/source/Plugins/ExpressionParser/Clang	*51*	*25*	*26*	49%
lldb/source/Plugins/Instruction/ARM	*4*	*2*	*2*	50%
lldb/source/Plugins/Instruction/ARM64	*2*	*0*	*2*	0%
lldb/source/Plugins/Instruction/MIPS	*2*	*0*	*2*	0%
lldb/source/Plugins/Instruction/MIPS64	*2*	*1*	*1*	50%
lldb/source/Plugins/Instruction/PPC64	*2*	*2*	*0*	100%
lldb/source/Plugins/InstrumentationRuntime/ASan	*2*	*2*	*0*	100%
lldb/source/Plugins/InstrumentationRuntime/MainThreadChecker	*2*	*2*	*0*	100%
lldb/source/Plugins/InstrumentationRuntime/TSan	*2*	*2*	*0*	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
lldb/source/Plugins/InstrumentationRuntime/UBSan	*2*	*2*	*0*	100%
lldb/source/Plugins/JITLoader/GDB	*2*	*1*	*1*	50%
lldb/source/Plugins/Language/ClangCommon	*2*	*2*	*0*	100%
lldb/source/Plugins/Language/CPlusPlus	*30*	*19*	*11*	63%
lldb/source/Plugins/Language/ObjC	*21*	*14*	*7*	66%
lldb/source/Plugins/Language/ObjCPlusPlus	*2*	*2*	*0*	100%
lldb/source/Plugins/LanguageRuntime/CPlusPlus	*2*	*0*	*2*	0%
lldb/source/Plugins/LanguageRuntime/CPlusPlus/ItaniumABI	*2*	*0*	*2*	0%
lldb/source/Plugins/LanguageRuntime/ObjC	*2*	*0*	*2*	0%
lldb/source/Plugins/LanguageRuntime/ObjC/AppleObjCRuntime	*16*	*5*	*11*	31%
lldb/source/Plugins/LanguageRuntime/RenderScript/RenderScriptRuntime	*8*	*3*	*5*	37%
lldb/source/Plugins/MemoryHistory/asan	*2*	*2*	*0*	100%
lldb/source/Plugins/ObjectContainer/BSD-Archive	*2*	*0*	*2*	0%
lldb/source/Plugins/ObjectContainer/Universal-Mach-O	*2*	*2*	*0*	100%
lldb/source/Plugins/ObjectFile/Breakpad	*4*	*3*	*1*	75%
lldb/source/Plugins/ObjectFile/ELF	*4*	*1*	*3*	25%
lldb/source/Plugins/ObjectFile/JIT	*2*	*0*	*2*	0%
lldb/source/Plugins/ObjectFile/Mach-O	*2*	*0*	*2*	0%
lldb/source/Plugins/ObjectFile/Minidump	*4*	*4*	*0*	100%
lldb/source/Plugins/ObjectFile/PDB	*2*	*2*	*0*	100%
lldb/source/Plugins/ObjectFile/PECOFF	*6*	*3*	*3*	50%
lldb/source/Plugins/ObjectFile/wasm	*2*	*2*	*0*	100%
lldb/source/Plugins/OperatingSystem/Python	*2*	*2*	*0*	100%
lldb/source/Plugins/Platform/Android	*6*	*3*	*3*	50%
lldb/source/Plugins/Platform/FreeBSD	*2*	*1*	*1*	50%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
lldb/source/Plugins/Platform/gdb-server	*2*	*1*	*1*	50%
lldb/source/Plugins/Platform/Linux	*2*	*1*	*1*	50%
lldb/source/Plugins/Platform/MacOSX	*20*	*11*	*9*	55%
lldb/source/Plugins/Platform/MacOSX/objcxx	*1*	*1*	*0*	100%
lldb/source/Plugins/Platform/NetBSD	*2*	*1*	*1*	50%
lldb/source/Plugins/Platform/OpenBSD	*2*	*1*	*1*	50%
lldb/source/Plugins/Platform/POSIX	*2*	*0*	*2*	0%
lldb/source/Plugins/Platform/QemuUser	*2*	*2*	*0*	100%
lldb/source/Plugins/Platform/Windows	*2*	*1*	*1*	50%
lldb/source/Plugins/Process/elf-core	*20*	*18*	*2*	90%
lldb/source/Plugins/Process/FreeBSD	*16*	*12*	*4*	75%
lldb/source/Plugins/Process/FreeBSDKernel	*10*	*8*	*2*	80%
lldb/source/Plugins/Process/gdb-remote	*26*	*15*	*11*	57%
lldb/source/Plugins/Process/Linux	*21*	*11*	*10*	52%
lldb/source/Plugins/Process/mach-core	*4*	*3*	*1*	75%
lldb/source/Plugins/Process/MacOSX-Kernel	*16*	*13*	*3*	81%
lldb/source/Plugins/Process/minidump	*17*	*10*	*7*	58%
lldb/source/Plugins/Process/NetBSD	*8*	*4*	*4*	50%
lldb/source/Plugins/Process/POSIX	*8*	*7*	*1*	87%
lldb/source/Plugins/Process/scripted	*4*	*4*	*0*	100%
lldb/source/Plugins/Process/Utility	*132*	*97*	*35*	73%
lldb/source/Plugins/Process/Windows/Common	*34*	*22*	*12*	64%
lldb/source/Plugins/Process/Windows/Common/arm	*2*	*1*	*1*	50%
lldb/source/Plugins/Process/Windows/Common/arm64	*2*	*1*	*1*	50%
lldb/source/Plugins/Process/Windows/Common/x64	*2*	*0*	*2*	0%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
lldb/source/Plugins/Process/Windows/Common/x86	*2*	*0*	*2*	0%
lldb/source/Plugins/REPL/Clang	*2*	*1*	*1*	50%
lldb/source/Plugins/ScriptInterpreter/Lua	*5*	*5*	*0*	100%
lldb/source/Plugins/ScriptInterpreter/None	*2*	*2*	*0*	100%
lldb/source/Plugins/ScriptInterpreter/Python	*16*	*12*	*4*	75%
lldb/source/Plugins/StructuredData/DarwinLog	*2*	*0*	*2*	0%
lldb/source/Plugins/SymbolFile/Breakpad	*2*	*0*	*2*	0%
lldb/source/Plugins/SymbolFile/DWARF	*65*	*39*	*26*	60%
lldb/source/Plugins/SymbolFile/NativePDB	*20*	*10*	*10*	50%
lldb/source/Plugins/SymbolFile/PDB	*6*	*4*	*2*	66%
lldb/source/Plugins/SymbolFile/Symtab	*2*	*2*	*0*	100%
lldb/source/Plugins/SymbolVendor/ELF	*2*	*2*	*0*	100%
lldb/source/Plugins/SymbolVendor/MacOSX	*2*	*2*	*0*	100%
lldb/source/Plugins/SymbolVendor/wasm	*2*	*2*	*0*	100%
lldb/source/Plugins/SystemRuntime/MacOSX	*10*	*1*	*9*	10%
lldb/source/Plugins/Trace/common	*8*	*7*	*1*	87%
lldb/source/Plugins/Trace/intel-pt	*18*	*17*	*1*	94%
lldb/source/Plugins/TraceExporter/common	*2*	*2*	*0*	100%
lldb/source/Plugins/TraceExporter/ctf	*4*	*3*	*1*	75%
lldb/source/Plugins/TypeSystem/Clang	*2*	*0*	*2*	0%
lldb/source/Plugins/UnwindAssembly/InstEmulation	*2*	*1*	*1*	50%
lldb/source/Plugins/UnwindAssembly/x86	*4*	*2*	*2*	50%
lldb/source/Symbol	*31*	*18*	*13*	58%
lldb/source/Target	*69*	*34*	*35*	49%
lldb/source/Utility	*58*	*46*	*12*	79%
lldb/source/Version	*1*	*1*	*0*	100%
lldb/tools/argdumper	*1*	*1*	*0*	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
lldb/tools/darwin-debug	*1*	*1*	*0*	100%
lldb/tools/debugserver/source	*51*	*40*	*11*	78%
lldb/tools/debugserver/source/MacOSX	*24*	*16*	*8*	66%
lldb/tools/debugserver/source/MacOSX/arm	*2*	*1*	*1*	50%
lldb/tools/debugserver/source/MacOSX/arm64	*2*	*1*	*1*	50%
lldb/tools/debugserver/source/MacOSX/i386	*3*	*0*	*3*	0%
lldb/tools/debugserver/source/MacOSX/x86_64	*3*	*0*	*3*	0%
lldb/tools/driver	*4*	*4*	*0*	100%
lldb/tools/intel-features	*1*	*1*	*0*	100%
lldb/tools/intel-features/intel-mpx	*2*	*1*	*1*	50%
lldb/tools/lldb-instr	*1*	*1*	*0*	100%
lldb/tools/lldb-server	*9*	*4*	*5*	44%
lldb/tools/lldb-test	*5*	*2*	*3*	40%
lldb/tools/lldb-vscode	*27*	*24*	*3*	88%
lldb/unittests	*1*	*1*	*0*	100%
lldb/unittests/API	*2*	*2*	*0*	100%
lldb/unittests/Breakpoint	*1*	*1*	*0*	100%
lldb/unittests/Core	*10*	*9*	*1*	90%
lldb/unittests/DataFormatter	*3*	*3*	*0*	100%
lldb/unittests/debugserver	*3*	*2*	*1*	66%
lldb/unittests/Disassembler	*2*	*0*	*2*	0%
lldb/unittests/Editline	*1*	*1*	*0*	100%
lldb/unittests/Expression	*5*	*3*	*2*	60%
lldb/unittests/Host	*16*	*11*	*5*	68%
lldb/unittests/Host/linux	*2*	*2*	*0*	100%
lldb/unittests/Host/posix	*1*	*0*	*1*	0%
lldb/unittests/Instruction	*1*	*0*	*1*	0%
lldb/unittests/Interpreter	*6*	*2*	*4*	33%
lldb/unittests/Language/CLanguages	*1*	*1*	*0*	100%
lldb/unittests/Language/CPlusPlus	*1*	*0*	*1*	0%
lldb/unittests/Language/Highlighting	*1*	*1*	*0*	100%
lldb/unittests/ObjectFile/Breakpad	*1*	*1*	*0*	100%
lldb/unittests/ObjectFile/ELF	*1*	*0*	*1*	0%
lldb/unittests/ObjectFile/MachO	*1*	*0*	*1*	0%
lldb/unittests/ObjectFile/PECOFF	*1*	*0*	*1*	0%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
lldb/unittests/Platform	3	2	1	66%
lldb/unittests/Platform/Android	1	0	1	0%
lldb/unittests/Process	1	1	0	100%
lldb/unittests/Process/gdb-remote	8	6	2	75%
lldb/unittests/Process/Linux	1	0	1	0%
lldb/unittests/Process/minidump	2	0	2	0%
lldb/unittests/Process/minidump/Inputs	1	1	0	100%
lldb/unittests/Process/POSIX	1	1	0	100%
lldb/unittests/Process/Utility	6	4	2	66%
lldb/unittests/ScriptInterpreter/Lua	2	2	0	100%
lldb/unittests/ScriptInterpreter/Python	3	2	1	66%
lldb/unittests/Signals	1	1	0	100%
lldb/unittests/Symbol	11	7	4	63%
lldb/unittests/SymbolFile/DWARF	6	4	2	66%
lldb/unittests/SymbolFile/DWARF/Inputs	1	1	0	100%
lldb/unittests/SymbolFile/NativePDB	1	1	0	100%
lldb/unittests/SymbolFile/PDB	1	0	1	0%
lldb/unittests/SymbolFile/PDB/Inputs	5	5	0	100%
lldb/unittests/Target	10	6	4	60%
lldb/unittests/TestingSupport	5	4	1	80%
lldb/unittests/TestingSupport/Host	1	1	0	100%
lldb/unittests/TestingSupport/Symbol	3	3	0	100%
lldb/unittests/Thread	1	1	0	100%
lldb/unittests/tools/lldb-server/inferior	2	0	2	0%
lldb/unittests/tools/lldb-server/tests	7	0	7	0%
lldb/unittests/UnwindAssembly/ARM64	1	0	1	0%
lldb/unittests/UnwindAssembly/PPC64	1	1	0	100%
lldb/unittests/UnwindAssembly/x86	1	0	1	0%
lldb/unittests/Utility	45	32	13	71%
lldb/utils/lit-cpuid	1	0	1	0%
lldb/utils/TableGen	6	6	0	100%
llvm/benchmarks	1	0	1	0%
llvm/bindings/go/llvm	6	3	3	50%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
llvm/bindings/ocaml/llvm	*1*	*1*	*0*	100%
llvm/cmake	*2*	*2*	*0*	100%
llvm/examples/BrainF	*3*	*0*	*3*	0%
llvm/examples/Bye	*1*	*1*	*0*	100%
llvm/examples/ExceptionDemo	*1*	*0*	*1*	0%
llvm/examples/Fibonacci	*1*	*0*	*1*	0%
llvm/examples/HowToUseJIT	*1*	*0*	*1*	0%
llvm/examples/HowToUseLLJIT	*1*	*1*	*0*	100%
llvm/examples/IRTransforms	*4*	*4*	*0*	100%
llvm/examples/Kaleidoscope/BuildingAJIT/Chapter1	*2*	*1*	*1*	50%
llvm/examples/Kaleidoscope/BuildingAJIT/Chapter2	*2*	*1*	*1*	50%
llvm/examples/Kaleidoscope/BuildingAJIT/Chapter3	*2*	*1*	*1*	50%
llvm/examples/Kaleidoscope/BuildingAJIT/Chapter4	*2*	*0*	*2*	0%
llvm/examples/Kaleidoscope/Chapter2	*1*	*1*	*0*	100%
llvm/examples/Kaleidoscope/Chapter3	*1*	*0*	*1*	0%
llvm/examples/Kaleidoscope/Chapter4	*1*	*0*	*1*	0%
llvm/examples/Kaleidoscope/Chapter5	*1*	*0*	*1*	0%
llvm/examples/Kaleidoscope/Chapter6	*1*	*0*	*1*	0%
llvm/examples/Kaleidoscope/Chapter7	*1*	*0*	*1*	0%
llvm/examples/Kaleidoscope/Chapter8	*1*	*0*	*1*	0%
llvm/examples/Kaleidoscope/Chapter9	*1*	*0*	*1*	0%
llvm/examples/Kaleidoscope/include	*1*	*1*	*0*	100%
llvm/examples/Kaleidoscope/MCJIT/cached	*2*	*0*	*2*	0%
llvm/examples/Kaleidoscope/MCJIT/complete	*1*	*0*	*1*	0%
llvm/examples/Kaleidoscope/MCJIT/initial	*1*	*0*	*1*	0%
llvm/examples/Kaleidoscope/MCJIT/lazy	*2*	*0*	*2*	0%
llvm/examples/ModuleMaker	*1*	*0*	*1*	0%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
llvm/examples/OrcV2Examples	*1*	*1*	*0*	100%
llvm/examples/OrcV2Examples/LLJITDumpObjects	*1*	*1*	*0*	100%
llvm/examples/OrcV2Examples/LLJITWithCustomObjectLinkingLayer	*1*	*1*	*0*	100%
llvm/examples/OrcV2Examples/LLJITWithExecutorProcessControl	*1*	*1*	*0*	100%
llvm/examples/OrcV2Examples/LLJITWithGDBRegistrationListener	*1*	*1*	*0*	100%
llvm/examples/OrcV2Examples/LLJITWithInitializers	*1*	*1*	*0*	100%
llvm/examples/OrcV2Examples/LLJITWithLazyReexports	*1*	*1*	*0*	100%
llvm/examples/OrcV2Examples/LLJITWithObjectCache	*1*	*1*	*0*	100%
llvm/examples/OrcV2Examples/LLJITWithObjectLinkingLayerPlugin	*1*	*0*	*1*	0%
llvm/examples/OrcV2Examples/LLJITWithOptimizingIRTransform	*1*	*1*	*0*	100%
llvm/examples/OrcV2Examples/LLJITWithRemoteDebugging	*3*	*1*	*2*	33%
llvm/examples/OrcV2Examples/LLJITWithThinLTOSummaries	*1*	*0*	*1*	0%
llvm/examples/ParallelJIT	*1*	*0*	*1*	0%
llvm/examples/SpeculativeJIT	*1*	*0*	*1*	0%
llvm/include/llvm	*8*	*2*	*6*	25%
llvm/include/llvm/ADT	*93*	*25*	*68*	26%
llvm/include/llvm/Analysis	*130*	*52*	*78*	40%
llvm/include/llvm/Analysis/Utils	*3*	*1*	*2*	33%
llvm/include/llvm/AsmParser	*5*	*2*	*3*	40%
llvm/include/llvm/BinaryFormat	*15*	*8*	*7*	53%
llvm/include/llvm/Bitcode	*7*	*2*	*5*	28%
llvm/include/llvm/Bitstream	*3*	*0*	*3*	0%
llvm/include/llvm/CodeGen	*158*	*51*	*107*	32%
llvm/include/llvm/CodeGen/GlobalISel	*27*	*8*	*19*	29%
llvm/include/llvm/CodeGen/MIRParser	*2*	*1*	*1*	50%
llvm/include/llvm/CodeGen/PBQP	*5*	*1*	*4*	20%
llvm/include/llvm/DebugInfo	*1*	*1*	*0*	100%
llvm/include/llvm/DebugInfo/CodeView	*57*	*40*	*17*	70%
llvm/include/llvm/DebugInfo/DWARF	*32*	*14*	*18*	43%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
llvm/include/llvm/DebugInfo/GSYM	*14*	*4*	*10*	28%
llvm/include/llvm/DebugInfo/MSF	*5*	*4*	*1*	80%
llvm/include/llvm/DebugInfo/PDB	*50*	*30*	*20*	60%
llvm/include/llvm/DebugInfo/PDB/DIA	*20*	*9*	*11*	45%
llvm/include/llvm/DebugInfo/PDB/Native	*54*	*35*	*19*	64%
llvm/include/llvm/DebugInfo/Symbolize	*5*	*3*	*2*	60%
llvm/include/llvm/Debuginfod	*3*	*3*	*0*	100%
llvm/include/llvm/Demangle	*7*	*3*	*4*	42%
llvm/include/llvm/DWARFLinker	*4*	*4*	*0*	100%
llvm/include/llvm/DWP	*3*	*3*	*0*	100%
llvm/include/llvm/ExecutionEngine	*12*	*2*	*10*	16%
llvm/include/llvm/ExecutionEngine/JITLink	*16*	*14*	*2*	87%
llvm/include/llvm/ExecutionEngine/Orc	*38*	*29*	*9*	76%
llvm/include/llvm/ExecutionEngine/Orc/Shared	*8*	*4*	*4*	50%
llvm/include/llvm/ExecutionEngine/Orc/TargetProcess	*7*	*7*	*0*	100%
llvm/include/llvm/FileCheck	*1*	*1*	*0*	100%
llvm/include/llvm/Frontend/OpenMP	*5*	*4*	*1*	80%
llvm/include/llvm/FuzzMutate	*6*	*0*	*6*	0%
llvm/include/llvm/InterfaceStub	*3*	*3*	*0*	100%
llvm/include/llvm/IR	*93*	*28*	*65*	30%
llvm/include/llvm/IRReader	*1*	*0*	*1*	0%
llvm/include/llvm/LineEditor	*1*	*0*	*1*	0%
llvm/include/llvm/Linker	*2*	*0*	*2*	0%
llvm/include/llvm/LTO	*4*	*1*	*3*	25%
llvm/include/llvm/LTO/legacy	*4*	*0*	*4*	0%
llvm/include/llvm/MC	*74*	*24*	*50*	32%
llvm/include/llvm/MC/MCDisassembler	*4*	*1*	*3*	25%
llvm/include/llvm/MC/MCParser	*8*	*3*	*5*	37%
llvm/include/llvm/MCA	*10*	*10*	*0*	100%
llvm/include/llvm/MCA/HardwareUnits	*6*	*4*	*2*	66%
llvm/include/llvm/MCA/Stages	*8*	*8*	*0*	100%
llvm/include/llvm/ObjCopy	*4*	*3*	*1*	75%
llvm/include/llvm/ObjCopy/COFF	*2*	*2*	*0*	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
llvm/include/llvm/ObjCopy/ELF	2	2	0	100%
llvm/include/llvm/ObjCopy/MachO	2	2	0	100%
llvm/include/llvm/ObjCopy/wasm	2	2	0	100%
llvm/include/llvm/ObjCopy/XCOFF	2	2	0	100%
llvm/include/llvm/Object	31	12	19	38%
llvm/include/llvm/ObjectYAML	16	12	4	75%
llvm/include/llvm/Option	5	1	4	20%
llvm/include/llvm/Passes	4	2	2	50%
llvm/include/llvm/ProfileData	11	5	6	45%
llvm/include/llvm/ProfileData/Coverage	3	2	1	66%
llvm/include/llvm/Remarks	12	11	1	91%
llvm/include/llvm/Support	186	68	118	36%
llvm/include/llvm/Support/FileSystem	1	1	0	100%
llvm/include/llvm/Support/Solaris/sys	1	1	0	100%
llvm/include/llvm/Support/Windows	1	0	1	0%
llvm/include/llvm/TableGen	9	3	6	33%
llvm/include/llvm/Target	6	2	4	33%
llvm/include/llvm/Testing/Support	3	2	1	66%
llvm/include/llvm/TextAPI	9	9	0	100%
llvm/include/llvm/ToolDrivers/llvm-dlltool	1	1	0	100%
llvm/include/llvm/ToolDrivers/llvm-lib	1	0	1	0%
llvm/include/llvm/Transforms	8	2	6	25%
llvm/include/llvm/Transforms/AggressiveInstCombine	1	0	1	0%
llvm/include/llvm/Transforms/Coroutines	4	4	0	100%
llvm/include/llvm/Transforms/InstCombine	2	1	1	50%
llvm/include/llvm/Transforms/Instrumentation	17	10	7	58%
llvm/include/llvm/Transforms/IPO	38	28	10	73%
llvm/include/llvm/Transforms/Scalar	75	47	28	62%
llvm/include/llvm/Transforms/Utils	74	44	30	59%
llvm/include/llvm/Transforms/Vectorize	5	1	4	20%
llvm/include/llvm/WindowsDriver	2	1	1	50%
llvm/include/llvm/WindowsManifest	1	1	0	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
llvm/include/llvm/WindowsResource	*3*	*1*	*2*	33%
llvm/include/llvm/XRay	*17*	*13*	*4*	76%
llvm/include/llvm-c	*27*	*12*	*15*	44%
llvm/include/llvm-c/Transforms	*9*	*3*	*6*	33%
llvm/lib/Analysis	*119*	*40*	*79*	33%
llvm/lib/AsmParser	*3*	*1*	*2*	33%
llvm/lib/BinaryFormat	*13*	*10*	*3*	76%
llvm/lib/Bitcode/Reader	*7*	*2*	*5*	28%
llvm/lib/Bitcode/Writer	*5*	*0*	*5*	0%
llvm/lib/Bitstream/Reader	*1*	*0*	*1*	0%
llvm/lib/CodeGen	*220*	*60*	*160*	27%
llvm/lib/CodeGen/AsmPrinter	*45*	*18*	*27*	40%
llvm/lib/CodeGen/GlobalISel	*24*	*9*	*15*	37%
llvm/lib/CodeGen/LiveDebugValues	*5*	*1*	*4*	20%
llvm/lib/CodeGen/MIRParser	*4*	*1*	*3*	25%
llvm/lib/CodeGen/SelectionDAG	*31*	*2*	*29*	6%
llvm/lib/DebugInfo/CodeView	*40*	*23*	*17*	57%
llvm/lib/DebugInfo/DWARF	*28*	*9*	*19*	32%
llvm/lib/DebugInfo/GSYM	*11*	*2*	*9*	18%
llvm/lib/DebugInfo/MSF	*4*	*3*	*1*	75%
llvm/lib/DebugInfo/PDB	*40*	*35*	*5*	87%
llvm/lib/DebugInfo/PDB/DIA	*18*	*15*	*3*	83%
llvm/lib/DebugInfo/PDB/Native	*50*	*37*	*13*	74%
llvm/lib/DebugInfo/Symbolize	*4*	*3*	*1*	75%
llvm/lib/Debuginfod	*3*	*3*	*0*	100%
llvm/lib/Demangle	*6*	*4*	*2*	66%
llvm/lib/DWARFLinker	*4*	*3*	*1*	75%
llvm/lib/DWP	*2*	*2*	*0*	100%
llvm/lib/ExecutionEngine	*5*	*1*	*4*	20%
llvm/lib/ExecutionEngine/IntelJITEvents	*5*	*0*	*5*	0%
llvm/lib/ExecutionEngine/Interpreter	*4*	*0*	*4*	0%
llvm/lib/ExecutionEngine/JITLink	*23*	*15*	*8*	65%
llvm/lib/ExecutionEngine/MCJIT	*2*	*0*	*2*	0%
llvm/lib/ExecutionEngine/OProfileJIT	*2*	*0*	*2*	0%
llvm/lib/ExecutionEngine/Orc	*37*	*22*	*15*	59%
llvm/lib/ExecutionEngine/Orc/Shared	*4*	*4*	*0*	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
llvm/lib/ExecutionEngine/Orc/TargetProcess	*8*	*7*	*1*	87%
llvm/lib/ExecutionEngine/PerfJITEvents	*1*	*0*	*1*	0%
llvm/lib/ExecutionEngine/RuntimeDyld	*12*	*1*	*11*	8%
llvm/lib/ExecutionEngine/RuntimeDyld/Targets	*10*	*1*	*9*	10%
llvm/lib/Extensions	*1*	*0*	*1*	0%
llvm/lib/FileCheck	*2*	*1*	*1*	50%
llvm/lib/Frontend/OpenACC	*1*	*1*	*0*	100%
llvm/lib/Frontend/OpenMP	*3*	*3*	*0*	100%
llvm/lib/FuzzMutate	*5*	*2*	*3*	40%
llvm/lib/InterfaceStub	*3*	*3*	*0*	100%
llvm/lib/IR	*69*	*20*	*49*	28%
llvm/lib/IRReader	*1*	*0*	*1*	0%
llvm/lib/LineEditor	*1*	*0*	*1*	0%
llvm/lib/Linker	*3*	*0*	*3*	0%
llvm/lib/LTO	*7*	*1*	*6*	14%
llvm/lib/MC	*65*	*21*	*44*	32%
llvm/lib/MC/MCDisassembler	*6*	*3*	*3*	50%
llvm/lib/MC/MCParser	*14*	*3*	*11*	21%
llvm/lib/MCA	*9*	*8*	*1*	88%
llvm/lib/MCA/HardwareUnits	*6*	*4*	*2*	66%
llvm/lib/MCA/Stages	*8*	*7*	*1*	87%
llvm/lib/ObjCopy	*4*	*3*	*1*	75%
llvm/lib/ObjCopy/COFF	*7*	*7*	*0*	100%
llvm/lib/ObjCopy/ELF	*3*	*3*	*0*	100%
llvm/lib/ObjCopy/MachO	*9*	*9*	*0*	100%
llvm/lib/ObjCopy/wasm	*7*	*7*	*0*	100%
llvm/lib/ObjCopy/XCOFF	*6*	*3*	*3*	50%
llvm/lib/Object	*31*	*16*	*15*	51%
llvm/lib/ObjectYAML	*23*	*9*	*14*	39%
llvm/lib/Option	*4*	*0*	*4*	0%
llvm/lib/Passes	*6*	*3*	*3*	50%
llvm/lib/ProfileData	*11*	*4*	*7*	36%
llvm/lib/ProfileData/Coverage	*3*	*0*	*3*	0%
llvm/lib/Remarks	*13*	*10*	*3*	76%
llvm/lib/Support	*144*	*61*	*83*	42%
llvm/lib/Support/Unix	*1*	*0*	*1*	0%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
llvm/lib/TableGen	*15*	*3*	*12*	20%
llvm/lib/Target	*5*	*1*	*4*	20%
llvm/lib/Target/AArch64	*60*	*7*	*53*	11%
llvm/lib/Target/AArch64/AsmParser	*1*	*0*	*1*	0%
llvm/lib/Target/AArch64/Disassembler	*4*	*1*	*3*	25%
llvm/lib/Target/AArch64/GISel	*14*	*3*	*11*	21%
llvm/lib/Target/AArch64/MCTargetDesc	*21*	*6*	*15*	28%
llvm/lib/Target/AArch64/TargetInfo	*2*	*1*	*1*	50%
llvm/lib/Target/AArch64/Utils	*2*	*0*	*2*	0%
llvm/lib/Target/AMDGPU	*169*	*38*	*131*	22%
llvm/lib/Target/AMDGPU/AsmParser	*1*	*0*	*1*	0%
llvm/lib/Target/AMDGPU/Disassembler	*2*	*0*	*2*	0%
llvm/lib/Target/AMDGPU/MCA	*2*	*2*	*0*	100%
llvm/lib/Target/AMDGPU/MCTargetDesc	*21*	*5*	*16*	23%
llvm/lib/Target/AMDGPU/TargetInfo	*2*	*1*	*1*	50%
llvm/lib/Target/AMDGPU/Utils	*11*	*4*	*7*	36%
llvm/lib/Target/ARC	*24*	*19*	*5*	79%
llvm/lib/Target/ARC/Disassembler	*1*	*0*	*1*	0%
llvm/lib/Target/ARC/MCTargetDesc	*7*	*6*	*1*	85%
llvm/lib/Target/ARC/TargetInfo	*2*	*2*	*0*	100%
llvm/lib/Target/ARM	*76*	*10*	*66*	13%
llvm/lib/Target/ARM/AsmParser	*1*	*0*	*1*	0%
llvm/lib/Target/ARM/Disassembler	*1*	*0*	*1*	0%
llvm/lib/Target/ARM/MCTargetDesc	*26*	*2*	*24*	7%
llvm/lib/Target/ARM/TargetInfo	*2*	*2*	*0*	100%
llvm/lib/Target/ARM/Utils	*2*	*0*	*2*	0%
llvm/lib/Target/AVR	*24*	*23*	*1*	95%
llvm/lib/Target/AVR/AsmParser	*1*	*1*	*0*	100%
llvm/lib/Target/AVR/Disassembler	*1*	*1*	*0*	100%
llvm/lib/Target/AVR/MCTargetDesc	*20*	*18*	*2*	90%
llvm/lib/Target/AVR/TargetInfo	*2*	*2*	*0*	100%
llvm/lib/Target/BPF	*32*	*9*	*23*	28%
llvm/lib/Target/BPF/AsmParser	*1*	*0*	*1*	0%
llvm/lib/Target/BPF/Disassembler	*1*	*0*	*1*	0%
llvm/lib/Target/BPF/MCTargetDesc	*8*	*1*	*7*	12%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
llvm/lib/Target/BPF/TargetInfo	*2*	*1*	*1*	50%
llvm/lib/Target/CSKY	*23*	*23*	*0*	100%
llvm/lib/Target/CSKY/AsmParser	*1*	*1*	*0*	100%
llvm/lib/Target/CSKY/Disassembler	*1*	*1*	*0*	100%
llvm/lib/Target/CSKY/MCTargetDesc	*15*	*14*	*1*	93%
llvm/lib/Target/CSKY/TargetInfo	*2*	*2*	*0*	100%
llvm/lib/Target/Hexagon	*80*	*6*	*74*	7%
llvm/lib/Target/Hexagon/AsmParser	*1*	*0*	*1*	0%
llvm/lib/Target/Hexagon/Disassembler	*1*	*0*	*1*	0%
llvm/lib/Target/Hexagon/MCTargetDesc	*26*	*6*	*20*	23%
llvm/lib/Target/Hexagon/TargetInfo	*2*	*1*	*1*	50%
llvm/lib/Target/Lanai	*28*	*20*	*8*	71%
llvm/lib/Target/Lanai/AsmParser	*1*	*0*	*1*	0%
llvm/lib/Target/Lanai/Disassembler	*2*	*2*	*0*	100%
llvm/lib/Target/Lanai/MCTargetDesc	*13*	*12*	*1*	92%
llvm/lib/Target/Lanai/TargetInfo	*2*	*2*	*0*	100%
llvm/lib/Target/LoongArch	*19*	*19*	*0*	100%
llvm/lib/Target/LoongArch/MCTargetDesc	*12*	*12*	*0*	100%
llvm/lib/Target/LoongArch/TargetInfo	*2*	*2*	*0*	100%
llvm/lib/Target/M68k	*26*	*25*	*1*	96%
llvm/lib/Target/M68k/AsmParser	*1*	*1*	*0*	100%
llvm/lib/Target/M68k/Disassembler	*1*	*1*	*0*	100%
llvm/lib/Target/M68k/GISel	*7*	*6*	*1*	85%
llvm/lib/Target/M68k/MCTargetDesc	*12*	*11*	*1*	91%
llvm/lib/Target/M68k/TargetInfo	*2*	*2*	*0*	100%
llvm/lib/Target/Mips	*70*	*12*	*58*	17%
llvm/lib/Target/Mips/AsmParser	*1*	*0*	*1*	0%
llvm/lib/Target/Mips/Disassembler	*1*	*0*	*1*	0%
llvm/lib/Target/Mips/MCTargetDesc	*25*	*6*	*19*	24%
llvm/lib/Target/Mips/TargetInfo	*2*	*2*	*0*	100%
llvm/lib/Target/MSP430	*20*	*0*	*20*	0%
llvm/lib/Target/MSP430/AsmParser	*1*	*0*	*1*	0%
llvm/lib/Target/MSP430/Disassembler	*1*	*0*	*1*	0%
llvm/lib/Target/MSP430/MCTargetDesc	*11*	*3*	*8*	27%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
llvm/lib/Target/MSP430/TargetInfo	*2*	*2*	*0*	100%
llvm/lib/Target/NVPTX	*44*	*10*	*34*	22%
llvm/lib/Target/NVPTX/MCTargetDesc	*9*	*6*	*3*	66%
llvm/lib/Target/NVPTX/TargetInfo	*2*	*2*	*0*	100%
llvm/lib/Target/PowerPC	*54*	*5*	*49*	9%
llvm/lib/Target/PowerPC/AsmParser	*1*	*0*	*1*	0%
llvm/lib/Target/PowerPC/Disassembler	*1*	*0*	*1*	0%
llvm/lib/Target/PowerPC/GISel	*7*	*7*	*0*	100%
llvm/lib/Target/PowerPC/MCTargetDesc	*20*	*5*	*15*	25%
llvm/lib/Target/PowerPC/TargetInfo	*2*	*2*	*0*	100%
llvm/lib/Target/RISCV	*36*	*17*	*19*	47%
llvm/lib/Target/RISCV/AsmParser	*1*	*0*	*1*	0%
llvm/lib/Target/RISCV/Disassembler	*1*	*0*	*1*	0%
llvm/lib/Target/RISCV/MCTargetDesc	*23*	*13*	*10*	56%
llvm/lib/Target/RISCV/TargetInfo	*2*	*2*	*0*	100%
llvm/lib/Target/Sparc	*23*	*3*	*20*	13%
llvm/lib/Target/Sparc/AsmParser	*1*	*0*	*1*	0%
llvm/lib/Target/Sparc/Disassembler	*1*	*0*	*1*	0%
llvm/lib/Target/Sparc/MCTargetDesc	*14*	*4*	*10*	28%
llvm/lib/Target/Sparc/TargetInfo	*2*	*2*	*0*	100%
llvm/lib/Target/SystemZ	*41*	*6*	*35*	14%
llvm/lib/Target/SystemZ/AsmParser	*1*	*0*	*1*	0%
llvm/lib/Target/SystemZ/Disassembler	*1*	*0*	*1*	0%
llvm/lib/Target/SystemZ/MCTargetDesc	*10*	*4*	*6*	40%
llvm/lib/Target/SystemZ/TargetInfo	*2*	*2*	*0*	100%
llvm/lib/Target/VE	*24*	*19*	*5*	79%
llvm/lib/Target/VE/AsmParser	*1*	*1*	*0*	100%
llvm/lib/Target/VE/Disassembler	*1*	*1*	*0*	100%
llvm/lib/Target/VE/MCTargetDesc	*14*	*14*	*0*	100%
llvm/lib/Target/VE/TargetInfo	*2*	*1*	*1*	50%
llvm/lib/Target/WebAssembly	*61*	*44*	*17*	72%
llvm/lib/Target/WebAssembly/AsmParser	*3*	*0*	*3*	0%
llvm/lib/Target/WebAssembly/Disassembler	*1*	*1*	*0*	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
llvm/lib/Target/WebAssembly/MCTargetDesc	*12*	*8*	*4*	66%
llvm/lib/Target/WebAssembly/TargetInfo	*2*	*2*	*0*	100%
llvm/lib/Target/WebAssembly/Utils	*4*	*4*	*0*	100%
llvm/lib/Target/X86	*82*	*19*	*63*	23%
llvm/lib/Target/X86/AsmParser	*3*	*0*	*3*	0%
llvm/lib/Target/X86/Disassembler	*2*	*0*	*2*	0%
llvm/lib/Target/X86/MCA	*2*	*2*	*0*	100%
llvm/lib/Target/X86/MCTargetDesc	*25*	*5*	*20*	20%
llvm/lib/Target/X86/TargetInfo	*2*	*1*	*1*	50%
llvm/lib/Target/XCore	*27*	*2*	*25*	7%
llvm/lib/Target/XCore/Disassembler	*1*	*0*	*1*	0%
llvm/lib/Target/XCore/MCTargetDesc	*6*	*3*	*3*	50%
llvm/lib/Target/XCore/TargetInfo	*2*	*1*	*1*	50%
llvm/lib/Testing/Support	*3*	*3*	*0*	100%
llvm/lib/TextAPI	*11*	*9*	*2*	81%
llvm/lib/ToolDrivers/llvm-dlltool	*1*	*0*	*1*	0%
llvm/lib/ToolDrivers/llvm-lib	*1*	*0*	*1*	0%
llvm/lib/Transforms/AggressiveInstCombine	*3*	*1*	*2*	33%
llvm/lib/Transforms/CFGuard	*1*	*1*	*0*	100%
llvm/lib/Transforms/Coroutines	*8*	*0*	*8*	0%
llvm/lib/Transforms/Hello	*1*	*0*	*1*	0%
llvm/lib/Transforms/InstCombine	*16*	*1*	*15*	6%
llvm/lib/Transforms/Instrumentation	*21*	*7*	*14*	33%
llvm/lib/Transforms/IPO	*44*	*9*	*35*	20%
llvm/lib/Transforms/ObjCARC	*15*	*4*	*11*	26%
llvm/lib/Transforms/Scalar	*79*	*16*	*63*	20%
llvm/lib/Transforms/Utils	*78*	*19*	*59*	24%
llvm/lib/Transforms/Vectorize	*22*	*13*	*9*	59%
llvm/lib/WindowsDriver	*1*	*1*	*0*	100%
llvm/lib/WindowsManifest	*1*	*1*	*0*	100%
llvm/lib/XRay	*14*	*11*	*3*	78%
llvm/tools/bugpoint	*12*	*1*	*11*	8%
llvm/tools/bugpoint-passes	*1*	*0*	*1*	0%
llvm/tools/dsymutil	*18*	*16*	*2*	88%
llvm/tools/gold	*1*	*0*	*1*	0%
llvm/tools/llc	*1*	*0*	*1*	0%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
llvm/tools/lli	*4*	*3*	*1*	75%
llvm/tools/lli/ChildTarget	*1*	*1*	*0*	100%
llvm/tools/llvm-ar	*1*	*0*	*1*	0%
llvm/tools/llvm-as	*1*	*0*	*1*	0%
llvm/tools/llvm-as-fuzzer	*1*	*1*	*0*	100%
llvm/tools/llvm-bcanalyzer	*1*	*1*	*0*	100%
llvm/tools/llvm-c-test	*2*	*0*	*2*	0%
llvm/tools/llvm-cat	*1*	*0*	*1*	0%
llvm/tools/llvm-cfi-verify	*1*	*0*	*1*	0%
llvm/tools/llvm-cfi-verify/lib	*4*	*1*	*3*	25%
llvm/tools/llvm-config	*1*	*0*	*1*	0%
llvm/tools/llvm-cov	*23*	*12*	*11*	52%
llvm/tools/llvm-cvtres	*1*	*0*	*1*	0%
llvm/tools/llvm-cxxdump	*4*	*1*	*3*	25%
llvm/tools/llvm-cxxfilt	*1*	*1*	*0*	100%
llvm/tools/llvm-cxxmap	*1*	*0*	*1*	0%
llvm/tools/llvm-debuginfod-find	*1*	*1*	*0*	100%
llvm/tools/llvm-diff	*1*	*0*	*1*	0%
llvm/tools/llvm-diff/lib	*6*	*0*	*6*	0%
llvm/tools/llvm-dis	*1*	*0*	*1*	0%
llvm/tools/llvm-dis-fuzzer	*1*	*1*	*0*	100%
llvm/tools/llvm-dlang-demangle-fuzzer	*2*	*2*	*0*	100%
llvm/tools/llvm-dwarfdump	*4*	*3*	*1*	75%
llvm/tools/llvm-dwarfdump/fuzzer	*1*	*0*	*1*	0%
llvm/tools/llvm-dwp	*1*	*0*	*1*	0%
llvm/tools/llvm-exegesis	*1*	*0*	*1*	0%
llvm/tools/llvm-exegesis/lib	*44*	*33*	*11*	75%
llvm/tools/llvm-exegesis/lib/AArch64	*1*	*1*	*0*	100%
llvm/tools/llvm-exegesis/lib/Mips	*1*	*0*	*1*	0%
llvm/tools/llvm-exegesis/lib/PowerPC	*1*	*1*	*0*	100%
llvm/tools/llvm-exegesis/lib/X86	*3*	*2*	*1*	66%
llvm/tools/llvm-extract	*1*	*0*	*1*	0%
llvm/tools/llvm-gsymutil	*1*	*1*	*0*	100%
llvm/tools/llvm-ifs	*3*	*2*	*1*	66%
llvm/tools/llvm-isel-fuzzer	*2*	*1*	*1*	50%
llvm/tools/llvm-itanium-demangle-fuzzer	*2*	*1*	*1*	50%
llvm/tools/llvm-jitlink	*4*	*2*	*2*	50%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
llvm/tools/llvm-jitlink/llvm-jitlink-executor	*1*	*1*	*0*	100%
llvm/tools/llvm-jitlistener	*1*	*0*	*1*	0%
llvm/tools/llvm-libtool-darwin	*1*	*1*	*0*	100%
llvm/tools/llvm-link	*1*	*1*	*0*	100%
llvm/tools/llvm-lipo	*1*	*0*	*1*	0%
llvm/tools/llvm-lto	*1*	*0*	*1*	0%
llvm/tools/llvm-lto2	*1*	*0*	*1*	0%
llvm/tools/llvm-mc	*3*	*1*	*2*	33%
llvm/tools/llvm-mc-assemble-fuzzer	*1*	*0*	*1*	0%
llvm/tools/llvm-mc-disassemble-fuzzer	*1*	*0*	*1*	0%
llvm/tools/llvm-mca	*7*	*7*	*0*	100%
llvm/tools/llvm-mca/Views	*20*	*19*	*1*	95%
llvm/tools/llvm-microsoft-demangle-fuzzer	*2*	*2*	*0*	100%
llvm/tools/llvm-ml	*3*	*1*	*2*	33%
llvm/tools/llvm-modextract	*1*	*1*	*0*	100%
llvm/tools/llvm-mt	*1*	*0*	*1*	0%
llvm/tools/llvm-nm	*1*	*0*	*1*	0%
llvm/tools/llvm-objcopy	*3*	*2*	*1*	66%
llvm/tools/llvm-objdump	*15*	*10*	*5*	66%
llvm/tools/llvm-opt-fuzzer	*2*	*0*	*2*	0%
llvm/tools/llvm-opt-report	*1*	*0*	*1*	0%
llvm/tools/llvm-pdbutil	*47*	*15*	*32*	31%
llvm/tools/llvm-profdata	*1*	*0*	*1*	0%
llvm/tools/llvm-profgen	*11*	*6*	*5*	54%
llvm/tools/llvm-rc	*12*	*6*	*6*	50%
llvm/tools/llvm-readobj	*19*	*3*	*16*	15%
llvm/tools/llvm-reduce	*7*	*6*	*1*	85%
llvm/tools/llvm-reduce/deltas	*40*	*39*	*1*	97%
llvm/tools/llvm-remark-size-diff	*1*	*1*	*0*	100%
llvm/tools/llvm-rtdyld	*1*	*0*	*1*	0%
llvm/tools/llvm-rust-demangle-fuzzer	*2*	*2*	*0*	100%
llvm/tools/llvm-shlib	*1*	*1*	*0*	100%
llvm/tools/llvm-sim	*1*	*0*	*1*	0%
llvm/tools/llvm-size	*1*	*0*	*1*	0%
llvm/tools/llvm-special-case-list-fuzzer	*2*	*2*	*0*	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
llvm/tools/llvm-split	*1*	*0*	*1*	0%
llvm/tools/llvm-stress	*1*	*0*	*1*	0%
llvm/tools/llvm-strings	*1*	*1*	*0*	100%
llvm/tools/llvm-symbolizer	*1*	*0*	*1*	0%
llvm/tools/llvm-tapi-diff	*3*	*3*	*0*	100%
llvm/tools/llvm-tli-checker	*1*	*0*	*1*	0%
llvm/tools/llvm-undname	*1*	*1*	*0*	100%
llvm/tools/llvm-xray	*19*	*15*	*4*	78%
llvm/tools/llvm-yaml-numeric-parser-fuzzer	*2*	*2*	*0*	100%
llvm/tools/llvm-yaml-parser-fuzzer	*2*	*2*	*0*	100%
llvm/tools/lto	*2*	*1*	*1*	50%
llvm/tools/obj2yaml	*10*	*5*	*5*	50%
llvm/tools/opt	*10*	*3*	*7*	30%
llvm/tools/remarks-shlib	*1*	*0*	*1*	0%
llvm/tools/sancov	*1*	*0*	*1*	0%
llvm/tools/sanstats	*1*	*1*	*0*	100%
llvm/tools/split-file	*1*	*0*	*1*	0%
llvm/tools/verify-uselistorder	*1*	*0*	*1*	0%
llvm/tools/vfabi-demangle-fuzzer	*1*	*1*	*0*	100%
llvm/tools/yaml2obj	*1*	*1*	*0*	100%
llvm/unittests/ADT	*77*	*29*	*48*	37%
llvm/unittests/Analysis	*38*	*13*	*25*	34%
llvm/unittests/AsmParser	*1*	*1*	*0*	100%
llvm/unittests/BinaryFormat	*6*	*5*	*1*	83%
llvm/unittests/Bitcode	*2*	*1*	*1*	50%
llvm/unittests/Bitstream	*2*	*1*	*1*	50%
llvm/unittests/CodeGen	*20*	*10*	*10*	50%
llvm/unittests/CodeGen/GlobalISel	*13*	*2*	*11*	15%
llvm/unittests/DebugInfo/CodeView	*4*	*2*	*2*	50%
llvm/unittests/DebugInfo/DWARF	*17*	*13*	*4*	76%
llvm/unittests/DebugInfo/GSYM	*1*	*0*	*1*	0%
llvm/unittests/DebugInfo/MSF	*3*	*2*	*1*	66%
llvm/unittests/DebugInfo/PDB	*5*	*3*	*2*	60%
llvm/unittests/DebugInfo/PDB/Inputs	*1*	*1*	*0*	100%
llvm/unittests/Debuginfod	*2*	*2*	*0*	100%
llvm/unittests/Demangle	*7*	*5*	*2*	71%
llvm/unittests/ExecutionEngine	*1*	*0*	*1*	0%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
llvm/unittests/ExecutionEngine/JITLink	*1*	*1*	*0*	100%
llvm/unittests/ExecutionEngine/MCJIT	*7*	*0*	*7*	0%
llvm/unittests/ExecutionEngine/Orc	*21*	*14*	*7*	66%
llvm/unittests/FileCheck	*1*	*0*	*1*	0%
llvm/unittests/Frontend	*4*	*3*	*1*	75%
llvm/unittests/FuzzMutate	*4*	*0*	*4*	0%
llvm/unittests/InterfaceStub	*1*	*1*	*0*	100%
llvm/unittests/IR	*36*	*6*	*30*	16%
llvm/unittests/LineEditor	*1*	*0*	*1*	0%
llvm/unittests/Linker	*1*	*0*	*1*	0%
llvm/unittests/MC	*7*	*4*	*3*	57%
llvm/unittests/MC/AMDGPU	*1*	*1*	*0*	100%
llvm/unittests/MC/SystemZ	*1*	*1*	*0*	100%
llvm/unittests/MI	*1*	*0*	*1*	0%
llvm/unittests/MIR	*1*	*0*	*1*	0%
llvm/unittests/ObjCopy	*1*	*1*	*0*	100%
llvm/unittests/Object	*9*	*6*	*3*	66%
llvm/unittests/ObjectYAML	*5*	*3*	*2*	60%
llvm/unittests/Option	*2*	*1*	*1*	50%
llvm/unittests/Passes	*5*	*5*	*0*	100%
llvm/unittests/ProfileData	*5*	*2*	*3*	40%
llvm/unittests/Remarks	*8*	*5*	*3*	62%
llvm/unittests/Support	*100*	*35*	*65*	35%
llvm/unittests/Support/CommandLineInit	*1*	*1*	*0*	100%
llvm/unittests/Support/DynamicLibrary	*4*	*0*	*4*	0%
llvm/unittests/TableGen	*3*	*1*	*2*	33%
llvm/unittests/Target/AArch64	*3*	*1*	*2*	33%
llvm/unittests/Target/AMDGPU	*2*	*2*	*0*	100%
llvm/unittests/Target/ARM	*2*	*1*	*1*	50%
llvm/unittests/Target/PowerPC	*1*	*1*	*0*	100%
llvm/unittests/Target/WebAssembly	*1*	*0*	*1*	0%
llvm/unittests/Target/X86	*1*	*0*	*1*	0%
llvm/unittests/Testing/Support	*1*	*1*	*0*	100%
llvm/unittests/TextAPI	*5*	*3*	*2*	60%
llvm/unittests/tools/llvm-cfi-verify	*2*	*1*	*1*	50%
llvm/unittests/tools/llvm-exegesis	*4*	*3*	*1*	75%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
llvm/unittests/tools/llvm-exegesis/AArch64	*1*	*1*	*0*	100%
llvm/unittests/tools/llvm-exegesis/ARM	*1*	*1*	*0*	100%
llvm/unittests/tools/llvm-exegesis/Common	*1*	*1*	*0*	100%
llvm/unittests/tools/llvm-exegesis/Mips	*5*	*3*	*2*	60%
llvm/unittests/tools/llvm-exegesis/PowerPC	*4*	*1*	*3*	25%
llvm/unittests/tools/llvm-exegesis/X86	*9*	*6*	*3*	66%
llvm/unittests/tools/llvm-profgen	*1*	*0*	*1*	0%
llvm/unittests/Transforms/IPO	*4*	*2*	*2*	50%
llvm/unittests/Transforms/Scalar	*2*	*0*	*2*	0%
llvm/unittests/Transforms/Utils	*19*	*8*	*11*	42%
llvm/unittests/Transforms/Vectorize	*7*	*7*	*0*	100%
llvm/unittests/XRay	*8*	*7*	*1*	87%
llvm/utils/FileCheck	*1*	*0*	*1*	0%
llvm/utils/fpcmp	*1*	*0*	*1*	0%
llvm/utils/KillTheDoctor	*1*	*0*	*1*	0%
llvm/utils/not	*1*	*1*	*0*	100%
llvm/utils/PerfectShuffle	*1*	*0*	*1*	0%
llvm/utils/TableGen	*78*	*13*	*65*	16%
llvm/utils/TableGen/GlobalISel	*17*	*10*	*7*	58%
llvm/utils/unittest/googlemock/include/gmock	*12*	*0*	*12*	0%
llvm/utils/unittest/googlemock/include/gmock/internal	*3*	*0*	*3*	0%
llvm/utils/unittest/googlemock/include/gmock/internal/custom	*3*	*0*	*3*	0%
llvm/utils/unittest/googletest/include/gtest	*11*	*0*	*11*	0%
llvm/utils/unittest/googletest/include/gtest/internal	*8*	*0*	*8*	0%
llvm/utils/unittest/googletest/include/gtest/internal/custom	*4*	*0*	*4*	0%
llvm/utils/unittest/googletest/src	*1*	*0*	*1*	0%
llvm/utils/unittest/UnitTestMain	*1*	*0*	*1*	0%
llvm/utils/yaml-bench	*1*	*0*	*1*	0%
mlir/examples/standalone/include/Standalone	*2*	*2*	*0*	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
mlir/examples/standalone/include/Standalone-c	*1*	*1*	*0*	100%
mlir/examples/standalone/lib/CAPI	*1*	*1*	*0*	100%
mlir/examples/standalone/lib/Standalone	*2*	*2*	*0*	100%
mlir/examples/standalone/python	*1*	*1*	*0*	100%
mlir/examples/standalone/standalone-opt	*1*	*1*	*0*	100%
mlir/examples/standalone/standalone-translate	*1*	*1*	*0*	100%
mlir/examples/toy/Ch1	*1*	*1*	*0*	100%
mlir/examples/toy/Ch1/include/toy	*3*	*3*	*0*	100%
mlir/examples/toy/Ch1/parser	*1*	*0*	*1*	0%
mlir/examples/toy/Ch2	*1*	*1*	*0*	100%
mlir/examples/toy/Ch2/include/toy	*5*	*5*	*0*	100%
mlir/examples/toy/Ch2/mlir	*2*	*2*	*0*	100%
mlir/examples/toy/Ch2/parser	*1*	*0*	*1*	0%
mlir/examples/toy/Ch3	*1*	*1*	*0*	100%
mlir/examples/toy/Ch3/include/toy	*5*	*5*	*0*	100%
mlir/examples/toy/Ch3/mlir	*3*	*3*	*0*	100%
mlir/examples/toy/Ch3/parser	*1*	*0*	*1*	0%
mlir/examples/toy/Ch4	*1*	*1*	*0*	100%
mlir/examples/toy/Ch4/include/toy	*7*	*7*	*0*	100%
mlir/examples/toy/Ch4/mlir	*4*	*4*	*0*	100%
mlir/examples/toy/Ch4/parser	*1*	*0*	*1*	0%
mlir/examples/toy/Ch5	*1*	*1*	*0*	100%
mlir/examples/toy/Ch5/include/toy	*7*	*7*	*0*	100%
mlir/examples/toy/Ch5/mlir	*5*	*5*	*0*	100%
mlir/examples/toy/Ch5/parser	*1*	*0*	*1*	0%
mlir/examples/toy/Ch6	*1*	*1*	*0*	100%
mlir/examples/toy/Ch6/include/toy	*7*	*7*	*0*	100%
mlir/examples/toy/Ch6/mlir	*6*	*6*	*0*	100%
mlir/examples/toy/Ch6/parser	*1*	*0*	*1*	0%
mlir/examples/toy/Ch7	*1*	*1*	*0*	100%
mlir/examples/toy/Ch7/include/toy	*7*	*7*	*0*	100%
mlir/examples/toy/Ch7/mlir	*6*	*6*	*0*	100%
mlir/examples/toy/Ch7/parser	*1*	*0*	*1*	0%
mlir/include/mlir	*5*	*5*	*0*	100%
mlir/include/mlir/Analysis	*7*	*5*	*2*	71%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
mlir/include/mlir/Analysis/AliasAnalysis	*1*	*1*	*0*	100%
mlir/include/mlir/Analysis/Presburger	*9*	*9*	*0*	100%
mlir/include/mlir/Bindings/Python	*1*	*0*	*1*	0%
mlir/include/mlir/CAPI	*12*	*12*	*0*	100%
mlir/include/mlir/Conversion	*1*	*1*	*0*	100%
mlir/include/mlir/Conversion/AffineToStandard	*1*	*1*	*0*	100%
mlir/include/mlir/Conversion/ArithmeticToLLVM	*1*	*1*	*0*	100%
mlir/include/mlir/Conversion/ArithmeticToSPIRV	*1*	*1*	*0*	100%
mlir/include/mlir/Conversion/ArmNeon2dToIntr	*1*	*1*	*0*	100%
mlir/include/mlir/Conversion/AsyncToLLVM	*1*	*1*	*0*	100%
mlir/include/mlir/Conversion/BufferizationToMemRef	*1*	*1*	*0*	100%
mlir/include/mlir/Conversion/ComplexToLLVM	*1*	*1*	*0*	100%
mlir/include/mlir/Conversion/ComplexToStandard	*1*	*1*	*0*	100%
mlir/include/mlir/Conversion/ControlFlowToLLVM	*1*	*1*	*0*	100%
mlir/include/mlir/Conversion/ControlFlowToSPIRV	*2*	*2*	*0*	100%
mlir/include/mlir/Conversion/FuncToSPIRV	*2*	*2*	*0*	100%
mlir/include/mlir/Conversion/GPUCommon	*1*	*1*	*0*	100%
mlir/include/mlir/Conversion/GPUToNVVM	*1*	*1*	*0*	100%
mlir/include/mlir/Conversion/GPUToROCDL	*2*	*2*	*0*	100%
mlir/include/mlir/Conversion/GPUToSPIRV	*2*	*2*	*0*	100%
mlir/include/mlir/Conversion/GPUToVulkan	*1*	*0*	*1*	0%
mlir/include/mlir/Conversion/LinalgToLLVM	*1*	*1*	*0*	100%
mlir/include/mlir/Conversion/LinalgToSPIRV	*2*	*2*	*0*	100%
mlir/include/mlir/Conversion/LinalgToStandard	*1*	*1*	*0*	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
mlir/include/mlir/Conversion/LLVMCommon	7	7	0	100%
mlir/include/mlir/Conversion/MathToLibm	1	1	0	100%
mlir/include/mlir/Conversion/MathToLLVM	1	1	0	100%
mlir/include/mlir/Conversion/MathToSPIRV	2	2	0	100%
mlir/include/mlir/Conversion/MemRefToLLVM	2	2	0	100%
mlir/include/mlir/Conversion/MemRefToSPIRV	2	2	0	100%
mlir/include/mlir/Conversion/OpenACCToLLVM	1	1	0	100%
mlir/include/mlir/Conversion/OpenACCToSCF	1	1	0	100%
mlir/include/mlir/Conversion/OpenMPToLLVM	1	1	0	100%
mlir/include/mlir/Conversion/PDLToPDLInterp	1	1	0	100%
mlir/include/mlir/Conversion/ReconcileUnrealizedCasts	1	1	0	100%
mlir/include/mlir/Conversion/SCFToControlFlow	1	1	0	100%
mlir/include/mlir/Conversion/SCFToGPU	2	2	0	100%
mlir/include/mlir/Conversion/SCFToOpenMP	1	1	0	100%
mlir/include/mlir/Conversion/SCFToSPIRV	2	2	0	100%
mlir/include/mlir/Conversion/ShapeToStandard	1	1	0	100%
mlir/include/mlir/Conversion/SPIRVToLLVM	2	2	0	100%
mlir/include/mlir/Conversion/StandardToLLVM	2	2	0	100%
mlir/include/mlir/Conversion/TensorToSPIRV	2	2	0	100%
mlir/include/mlir/Conversion/TosaToLinalg	1	1	0	100%
mlir/include/mlir/Conversion/TosaToSCF	1	1	0	100%
mlir/include/mlir/Conversion/TosaToStandard	1	1	0	100%
mlir/include/mlir/Conversion/VectorToGPU	1	1	0	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
mlir/include/mlir/Conversion/VectorToLLVM	*1*	*1*	*0*	100%
mlir/include/mlir/Conversion/VectorToROCDL	*1*	*1*	*0*	100%
mlir/include/mlir/Conversion/VectorToSCF	*1*	*1*	*0*	100%
mlir/include/mlir/Conversion/VectorToSPIRV	*2*	*2*	*0*	100%
mlir/include/mlir/Dialect	*2*	*2*	*0*	100%
mlir/include/mlir/Dialect/Affine	*4*	*4*	*0*	100%
mlir/include/mlir/Dialect/Affine/Analysis	*5*	*5*	*0*	100%
mlir/include/mlir/Dialect/Affine/IR	*3*	*3*	*0*	100%
mlir/include/mlir/Dialect/AMX	*2*	*2*	*0*	100%
mlir/include/mlir/Dialect/Arithmetic/IR	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/Arithmetic/Transforms	*2*	*2*	*0*	100%
mlir/include/mlir/Dialect/Arithmetic/Utils	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/ArmNeon	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/ArmSVE	*2*	*2*	*0*	100%
mlir/include/mlir/Dialect/Async	*2*	*2*	*0*	100%
mlir/include/mlir/Dialect/Async/IR	*2*	*2*	*0*	100%
mlir/include/mlir/Dialect/Bufferization/IR	*3*	*3*	*0*	100%
mlir/include/mlir/Dialect/Bufferization/Transforms	*4*	*4*	*0*	100%
mlir/include/mlir/Dialect/Complex/IR	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/ControlFlow/IR	*2*	*2*	*0*	100%
mlir/include/mlir/Dialect/DLTI	*2*	*2*	*0*	100%
mlir/include/mlir/Dialect/EmitC/IR	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/Func/IR	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/Func/Transforms	*3*	*3*	*0*	100%
mlir/include/mlir/Dialect/GPU	*5*	*5*	*0*	100%
mlir/include/mlir/Dialect/Linalg	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/Linalg/Analysis	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/Linalg/ComprehensiveBufferize	*2*	*2*	*0*	100%
mlir/include/mlir/Dialect/Linalg/IR	*2*	*2*	*0*	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
mlir/include/mlir/Dialect/Linalg/Transforms	*5*	*5*	*0*	100%
mlir/include/mlir/Dialect/Linalg/Utils	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/LLVMIR	*5*	*5*	*0*	100%
mlir/include/mlir/Dialect/LLVMIR/Transforms	*2*	*2*	*0*	100%
mlir/include/mlir/Dialect/Math/IR	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/Math/Transforms	*2*	*2*	*0*	100%
mlir/include/mlir/Dialect/MemRef/IR	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/MemRef/Transforms	*2*	*2*	*0*	100%
mlir/include/mlir/Dialect/MemRef/Utils	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/OpenACC	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/OpenMP	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/PDL/IR	*3*	*3*	*0*	100%
mlir/include/mlir/Dialect/PDLInterp/IR	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/Quant	*6*	*6*	*0*	100%
mlir/include/mlir/Dialect/SCF	*4*	*4*	*0*	100%
mlir/include/mlir/Dialect/SCF/Utils	*2*	*2*	*0*	100%
mlir/include/mlir/Dialect/Shape/IR	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/Shape/Transforms	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/SparseTensor/IR	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/SparseTensor/Pipelines	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/SparseTensor/Transforms	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/SparseTensor/Utils	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/SPIRV/IR	*9*	*9*	*0*	100%
mlir/include/mlir/Dialect/SPIRV/Linking	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/SPIRV/Transforms	*2*	*2*	*0*	100%
mlir/include/mlir/Dialect/SPIRV/Utils	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/Tensor/IR	*3*	*3*	*0*	100%
mlir/include/mlir/Dialect/Tensor/Transforms	*3*	*3*	*0*	100%
mlir/include/mlir/Dialect/Tensor/Utils	*1*	*1*	*0*	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
mlir/include/mlir/Dialect/Tosa/IR	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/Tosa/Transforms	*2*	*2*	*0*	100%
mlir/include/mlir/Dialect/Tosa/Utils	*3*	*3*	*0*	100%
mlir/include/mlir/Dialect/Utils	*4*	*4*	*0*	100%
mlir/include/mlir/Dialect/Vector/IR	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/Vector/Transforms	*4*	*4*	*0*	100%
mlir/include/mlir/Dialect/Vector/Utils	*1*	*1*	*0*	100%
mlir/include/mlir/Dialect/X86Vector	*2*	*2*	*0*	100%
mlir/include/mlir/ExecutionEngine	*8*	*7*	*1*	87%
mlir/include/mlir/Interfaces	*14*	*13*	*1*	92%
mlir/include/mlir/IR	*49*	*29*	*20*	59%
mlir/include/mlir/Parser	*1*	*1*	*0*	100%
mlir/include/mlir/Pass	*6*	*0*	*6*	0%
mlir/include/mlir/Reducer	*5*	*5*	*0*	100%
mlir/include/mlir/Rewrite	*2*	*2*	*0*	100%
mlir/include/mlir/Support	*15*	*9*	*6*	60%
mlir/include/mlir/TableGen	*21*	*19*	*2*	90%
mlir/include/mlir/Target/Cpp	*1*	*1*	*0*	100%
mlir/include/mlir/Target/LLVMIR	*6*	*6*	*0*	100%
mlir/include/mlir/Target/LLVMIR/Dialect	*1*	*1*	*0*	100%
mlir/include/mlir/Target/LLVMIR/Dialect/AMX	*1*	*1*	*0*	100%
mlir/include/mlir/Target/LLVMIR/Dialect/ArmNeon	*1*	*1*	*0*	100%
mlir/include/mlir/Target/LLVMIR/Dialect/ArmSVE	*1*	*1*	*0*	100%
mlir/include/mlir/Target/LLVMIR/Dialect/LLVMIR	*1*	*1*	*0*	100%
mlir/include/mlir/Target/LLVMIR/Dialect/NVVM	*1*	*1*	*0*	100%
mlir/include/mlir/Target/LLVMIR/Dialect/OpenACC	*1*	*1*	*0*	100%
mlir/include/mlir/Target/LLVMIR/Dialect/OpenMP	*1*	*1*	*0*	100%
mlir/include/mlir/Target/LLVMIR/Dialect/ROCDL	*1*	*1*	*0*	100%
mlir/include/mlir/Target/LLVMIR/Dialect/X86Vector	*1*	*1*	*0*	100%
mlir/include/mlir/Target/SPIRV	*3*	*3*	*0*	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
mlir/include/mlir/Tools/mlir-lsp-server	*1*	*1*	*0*	100%
mlir/include/mlir/Tools/mlir-reduce	*1*	*1*	*0*	100%
mlir/include/mlir/Tools/PDLL/AST	*4*	*2*	*2*	50%
mlir/include/mlir/Tools/PDLL/CodeGen	*2*	*2*	*0*	100%
mlir/include/mlir/Tools/PDLL/ODS	*4*	*4*	*0*	100%
mlir/include/mlir/Tools/PDLL/Parser	*1*	*1*	*0*	100%
mlir/include/mlir/Transforms	*9*	*7*	*2*	77%
mlir/include/mlir-c	*15*	*15*	*0*	100%
mlir/include/mlir-c/Bindings/Python	*1*	*1*	*0*	100%
mlir/include/mlir-c/Dialect	*11*	*11*	*0*	100%
mlir/lib/Analysis	*7*	*7*	*0*	100%
mlir/lib/Analysis/AliasAnalysis	*1*	*1*	*0*	100%
mlir/lib/Analysis/Presburger	*8*	*8*	*0*	100%
mlir/lib/Bindings/Python	*23*	*23*	*0*	100%
mlir/lib/Bindings/Python/Conversions	*1*	*1*	*0*	100%
mlir/lib/Bindings/Python/Transforms	*1*	*1*	*0*	100%
mlir/lib/CAPI/Conversion	*1*	*1*	*0*	100%
mlir/lib/CAPI/Debug	*1*	*1*	*0*	100%
mlir/lib/CAPI/Dialect	*15*	*15*	*0*	100%
mlir/lib/CAPI/ExecutionEngine	*1*	*1*	*0*	100%
mlir/lib/CAPI/Interfaces	*1*	*1*	*0*	100%
mlir/lib/CAPI/IR	*10*	*10*	*0*	100%
mlir/lib/CAPI/Registration	*1*	*1*	*0*	100%
mlir/lib/CAPI/Transforms	*1*	*1*	*0*	100%
mlir/lib/Conversion	*1*	*1*	*0*	100%
mlir/lib/Conversion/AffineToStandard	*1*	*1*	*0*	100%
mlir/lib/Conversion/ArithmeticToLLVM	*1*	*1*	*0*	100%
mlir/lib/Conversion/ArithmeticToSPIRV	*1*	*1*	*0*	100%
mlir/lib/Conversion/ArmNeon2dToIntr	*1*	*1*	*0*	100%
mlir/lib/Conversion/AsyncToLLVM	*1*	*1*	*0*	100%
mlir/lib/Conversion/BufferizationToMemRef	*1*	*0*	*1*	0%
mlir/lib/Conversion/ComplexToLLVM	*1*	*1*	*0*	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
mlir/lib/Conversion/ComplexToStandard	1	1	0	100%
mlir/lib/Conversion/ControlFlowToLLVM	1	1	0	100%
mlir/lib/Conversion/ControlFlowToSPIRV	2	2	0	100%
mlir/lib/Conversion/FuncToSPIRV	2	2	0	100%
mlir/lib/Conversion/GPUCommon	5	4	1	80%
mlir/lib/Conversion/GPUToNVVM	2	2	0	100%
mlir/lib/Conversion/GPUToROCDL	1	1	0	100%
mlir/lib/Conversion/GPUToSPIRV	2	2	0	100%
mlir/lib/Conversion/GPUToVulkan	2	2	0	100%
mlir/lib/Conversion/LinalgToLLVM	1	1	0	100%
mlir/lib/Conversion/LinalgToSPIRV	2	1	1	50%
mlir/lib/Conversion/LinalgToStandard	1	0	1	0%
mlir/lib/Conversion/LLVMCommon	8	8	0	100%
mlir/lib/Conversion/MathToLibm	1	1	0	100%
mlir/lib/Conversion/MathToLLVM	1	1	0	100%
mlir/lib/Conversion/MathToSPIRV	2	2	0	100%
mlir/lib/Conversion/MemRefToLLVM	2	2	0	100%
mlir/lib/Conversion/MemRefToSPIRV	2	2	0	100%
mlir/lib/Conversion/OpenACCToLLVM	1	1	0	100%
mlir/lib/Conversion/OpenACCToSCF	1	1	0	100%
mlir/lib/Conversion/OpenMPToLLVM	1	1	0	100%
mlir/lib/Conversion/PDLToPDLInterp	7	7	0	100%
mlir/lib/Conversion/ReconcileUnrealizedCasts	1	1	0	100%
mlir/lib/Conversion/SCFToControlFlow	1	1	0	100%
mlir/lib/Conversion/SCFToGPU	2	2	0	100%
mlir/lib/Conversion/SCFToOpenMP	1	1	0	100%
mlir/lib/Conversion/SCFToSPIRV	2	2	0	100%
mlir/lib/Conversion/ShapeToStandard	2	2	0	100%
mlir/lib/Conversion/SPIRVCommon	1	1	0	100%
mlir/lib/Conversion/SPIRVToLLVM	3	3	0	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
mlir/lib/Conversion/StandardToLLVM	1	1	0	100%
mlir/lib/Conversion/TensorToSPIRV	2	2	0	100%
mlir/lib/Conversion/TosaToLinalg	4	4	0	100%
mlir/lib/Conversion/TosaToSCF	2	2	0	100%
mlir/lib/Conversion/TosaToStandard	2	2	0	100%
mlir/lib/Conversion/VectorToGPU	1	0	1	0%
mlir/lib/Conversion/VectorToLLVM	2	2	0	100%
mlir/lib/Conversion/VectorToROCDL	1	1	0	100%
mlir/lib/Conversion/VectorToSCF	1	1	0	100%
mlir/lib/Conversion/VectorToSPIRV	2	1	1	50%
mlir/lib/Dialect	1	1	0	100%
mlir/lib/Dialect/Affine/Analysis	5	5	0	100%
mlir/lib/Dialect/Affine/IR	3	2	1	66%
mlir/lib/Dialect/Affine/Transforms	14	14	0	100%
mlir/lib/Dialect/Affine/Utils	3	3	0	100%
mlir/lib/Dialect/AMX/IR	1	1	0	100%
mlir/lib/Dialect/AMX/Transforms	1	1	0	100%
mlir/lib/Dialect/Arithmetic/IR	2	1	1	50%
mlir/lib/Dialect/Arithmetic/Transforms	4	3	1	75%
mlir/lib/Dialect/Arithmetic/Utils	1	1	0	100%
mlir/lib/Dialect/ArmNeon/IR	1	1	0	100%
mlir/lib/Dialect/ArmSVE/IR	1	1	0	100%
mlir/lib/Dialect/ArmSVE/Transforms	1	1	0	100%
mlir/lib/Dialect/Async/IR	1	1	0	100%
mlir/lib/Dialect/Async/Transforms	6	6	0	100%
mlir/lib/Dialect/Bufferization/IR	4	4	0	100%
mlir/lib/Dialect/Bufferization/Transforms	7	7	0	100%
mlir/lib/Dialect/Complex/IR	2	2	0	100%
mlir/lib/Dialect/ControlFlow/IR	1	1	0	100%
mlir/lib/Dialect/DLTI	2	2	0	100%
mlir/lib/Dialect/EmitC/IR	1	1	0	100%
mlir/lib/Dialect/Func/IR	1	1	0	100%
mlir/lib/Dialect/Func/Transforms	4	4	0	100%
mlir/lib/Dialect/GPU/IR	1	1	0	100%
mlir/lib/Dialect/GPU/Transforms	9	7	2	77%
mlir/lib/Dialect/Linalg/Analysis	1	1	0	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
mlir/lib/Dialect/Linalg/ComprehensiveBufferize	2	2	0	100%
mlir/lib/Dialect/Linalg/IR	3	3	0	100%
mlir/lib/Dialect/Linalg/Transforms	25	25	0	100%
mlir/lib/Dialect/Linalg/Utils	1	1	0	100%
mlir/lib/Dialect/LLVMIR/IR	7	5	2	71%
mlir/lib/Dialect/LLVMIR/Transforms	2	2	0	100%
mlir/lib/Dialect/Math/IR	2	2	0	100%
mlir/lib/Dialect/Math/Transforms	3	3	0	100%
mlir/lib/Dialect/MemRef/IR	2	2	0	100%
mlir/lib/Dialect/MemRef/Transforms	7	6	1	85%
mlir/lib/Dialect/MemRef/Utils	1	1	0	100%
mlir/lib/Dialect/OpenACC/IR	1	1	0	100%
mlir/lib/Dialect/OpenMP/IR	1	1	0	100%
mlir/lib/Dialect/PDL/IR	2	2	0	100%
mlir/lib/Dialect/PDLInterp/IR	1	1	0	100%
mlir/lib/Dialect/Quant/IR	4	4	0	100%
mlir/lib/Dialect/Quant/Transforms	3	3	0	100%
mlir/lib/Dialect/Quant/Utils	3	3	0	100%
mlir/lib/Dialect/SCF	1	1	0	100%
mlir/lib/Dialect/SCF/Transforms	12	11	1	91%
mlir/lib/Dialect/SCF/Utils	2	2	0	100%
mlir/lib/Dialect/Shape/IR	1	1	0	100%
mlir/lib/Dialect/Shape/Transforms	5	5	0	100%
mlir/lib/Dialect/SparseTensor/IR	1	1	0	100%
mlir/lib/Dialect/SparseTensor/Pipelines	1	1	0	100%
mlir/lib/Dialect/SparseTensor/Transforms	5	4	1	80%
mlir/lib/Dialect/SparseTensor/Utils	1	1	0	100%
mlir/lib/Dialect/SPIRV/IR	8	6	2	75%
mlir/lib/Dialect/SPIRV/Linking/ModuleCombiner	1	1	0	100%
mlir/lib/Dialect/SPIRV/Transforms	7	6	1	85%
mlir/lib/Dialect/SPIRV/Utils	1	1	0	100%
mlir/lib/Dialect/Tensor/IR	4	4	0	100%
mlir/lib/Dialect/Tensor/Transforms	4	4	0	100%
mlir/lib/Dialect/Tensor/Utils	1	1	0	100%
mlir/lib/Dialect/Tosa/IR	1	1	0	100%
mlir/lib/Dialect/Tosa/Transforms	6	6	0	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
mlir/lib/Dialect/Tosa/Utils	*2*	*2*	*0*	100%
mlir/lib/Dialect/Utils	*4*	*4*	*0*	100%
mlir/lib/Dialect/Vector/IR	*1*	*0*	*1*	0%
mlir/lib/Dialect/Vector/Transforms	*11*	*11*	*0*	100%
mlir/lib/Dialect/Vector/Utils	*1*	*1*	*0*	100%
mlir/lib/Dialect/X86Vector/IR	*1*	*1*	*0*	100%
mlir/lib/Dialect/X86Vector/Transforms	*2*	*2*	*0*	100%
mlir/lib/ExecutionEngine	*9*	*9*	*0*	100%
mlir/lib/Interfaces	*12*	*12*	*0*	100%
mlir/lib/IR	*38*	*31*	*7*	81%
mlir/lib/Parser	*14*	*10*	*4*	71%
mlir/lib/Pass	*8*	*6*	*2*	75%
mlir/lib/Reducer	*4*	*4*	*0*	100%
mlir/lib/Rewrite	*4*	*3*	*1*	75%
mlir/lib/Support	*8*	*8*	*0*	100%
mlir/lib/TableGen	*18*	*18*	*0*	100%
mlir/lib/Target/Cpp	*2*	*2*	*0*	100%
mlir/lib/Target/LLVMIR	*7*	*6*	*1*	85%
mlir/lib/Target/LLVMIR/Dialect/AMX	*1*	*1*	*0*	100%
mlir/lib/Target/LLVMIR/Dialect/ArmNeon	*1*	*1*	*0*	100%
mlir/lib/Target/LLVMIR/Dialect/ArmSVE	*1*	*1*	*0*	100%
mlir/lib/Target/LLVMIR/Dialect/LLVMIR	*1*	*1*	*0*	100%
mlir/lib/Target/LLVMIR/Dialect/NVVM	*1*	*1*	*0*	100%
mlir/lib/Target/LLVMIR/Dialect/OpenACC	*1*	*0*	*1*	0%
mlir/lib/Target/LLVMIR/Dialect/OpenMP	*1*	*1*	*0*	100%
mlir/lib/Target/LLVMIR/Dialect/ROCDL	*1*	*1*	*0*	100%
mlir/lib/Target/LLVMIR/Dialect/X86Vector	*1*	*1*	*0*	100%
mlir/lib/Target/SPIRV	*2*	*2*	*0*	100%
mlir/lib/Target/SPIRV/Deserialization	*4*	*3*	*1*	75%
mlir/lib/Target/SPIRV/Serialization	*4*	*3*	*1*	75%
mlir/lib/Tools/mlir-lsp-server	*5*	*4*	*1*	80%
mlir/lib/Tools/mlir-lsp-server/lsp	*6*	*4*	*2*	66%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
mlir/lib/Tools/mlir-reduce	*1*	*1*	*0*	100%
mlir/lib/Tools/PDLL/AST	*6*	*5*	*1*	83%
mlir/lib/Tools/PDLL/CodeGen	*2*	*1*	*1*	50%
mlir/lib/Tools/PDLL/ODS	*3*	*3*	*0*	100%
mlir/lib/Tools/PDLL/Parser	*3*	*1*	*2*	33%
mlir/lib/Transforms	*13*	*11*	*2*	84%
mlir/lib/Transforms/Utils	*6*	*6*	*0*	100%
mlir/lib/Translation	*1*	*1*	*0*	100%
mlir/tools/mlir-cpu-runner	*1*	*1*	*0*	100%
mlir/tools/mlir-linalg-ods-gen	*1*	*1*	*0*	100%
mlir/tools/mlir-lsp-server	*1*	*1*	*0*	100%
mlir/tools/mlir-opt	*1*	*1*	*0*	100%
mlir/tools/mlir-pdll	*1*	*1*	*0*	100%
mlir/tools/mlir-reduce	*1*	*1*	*0*	100%
mlir/tools/mlir-shlib	*1*	*1*	*0*	100%
mlir/tools/mlir-spirv-cpu-runner	*1*	*1*	*0*	100%
mlir/tools/mlir-tblgen	*29*	*28*	*1*	96%
mlir/tools/mlir-translate	*1*	*1*	*0*	100%
mlir/tools/mlir-vulkan-runner	*4*	*4*	*0*	100%
mlir/unittests/Analysis/Presburger	*8*	*8*	*0*	100%
mlir/unittests/Conversion/PDLToPDLInterp	*1*	*1*	*0*	100%
mlir/unittests/Dialect	*1*	*1*	*0*	100%
mlir/unittests/Dialect/Affine/Analysis	*3*	*3*	*0*	100%
mlir/unittests/Dialect/Quant	*1*	*1*	*0*	100%
mlir/unittests/Dialect/SparseTensor	*1*	*1*	*0*	100%
mlir/unittests/Dialect/SPIRV	*2*	*2*	*0*	100%
mlir/unittests/Dialect/Utils	*1*	*1*	*0*	100%
mlir/unittests/ExecutionEngine	*1*	*1*	*0*	100%
mlir/unittests/Interfaces	*3*	*3*	*0*	100%
mlir/unittests/IR	*7*	*7*	*0*	100%
mlir/unittests/Pass	*3*	*3*	*0*	100%
mlir/unittests/Rewrite	*1*	*1*	*0*	100%
mlir/unittests/Support	*5*	*4*	*1*	80%
mlir/unittests/TableGen	*5*	*3*	*2*	60%
mlir/unittests/Transforms	*2*	*2*	*0*	100%
openmp/libompd/src	*9*	*9*	*0*	100%
openmp/libomptarget/DeviceRTL/include	*8*	*8*	*0*	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
openmp/libomptarget/DeviceRTL/src	*12*	*9*	*3*	75%
openmp/libomptarget/include	*9*	*8*	*1*	88%
openmp/libomptarget/plugins/amdgpu/dynamic_hsa	*3*	*2*	*1*	66%
openmp/libomptarget/plugins/amdgpu/impl	*13*	*10*	*3*	76%
openmp/libomptarget/plugins/amdgpu/src	*2*	*1*	*1*	50%
openmp/libomptarget/plugins/common/elf_common	*2*	*2*	*0*	100%
openmp/libomptarget/plugins/common/MemoryManager	*1*	*1*	*0*	100%
openmp/libomptarget/plugins/cuda/dynamic_cuda	*2*	*2*	*0*	100%
openmp/libomptarget/plugins/cuda/src	*1*	*0*	*1*	0%
openmp/libomptarget/plugins/generic-elf-64bit/src	*1*	*1*	*0*	100%
openmp/libomptarget/plugins/remote/include	*1*	*1*	*0*	100%
openmp/libomptarget/plugins/remote/lib	*1*	*0*	*1*	0%
openmp/libomptarget/plugins/remote/server	*3*	*3*	*0*	100%
openmp/libomptarget/plugins/remote/src	*3*	*2*	*1*	66%
openmp/libomptarget/plugins/ve/src	*1*	*1*	*0*	100%
openmp/libomptarget/src	*7*	*6*	*1*	85%
openmp/libomptarget/tools/deviceinfo	*1*	*1*	*0*	100%
openmp/runtime/doc/doxygen	*1*	*1*	*0*	100%
openmp/runtime/src	*75*	*65*	*10*	86%
openmp/runtime/src/thirdparty/ittnotify	*6*	*5*	*1*	83%
openmp/runtime/src/thirdparty/ittnotify/legacy	*1*	*1*	*0*	100%
openmp/tools/archer	*1*	*1*	*0*	100%
openmp/tools/archer/tests/ompt	*1*	*1*	*0*	100%
openmp/tools/multiplex	*1*	*1*	*0*	100%
openmp/tools/multiplex/tests	*1*	*1*	*0*	100%
openmp/tools/multiplex/tests/custom_data_storage	*2*	*1*	*1*	50%
openmp/tools/multiplex/tests/print	*2*	*2*	*0*	100%

Directory	Total Files	Formatted Files	Unformatted Files	% Complete
polly/include/polly	*25*	*25*	*0*	100%
polly/include/polly/CodeGen	*14*	*14*	*0*	100%
polly/include/polly/Support	*12*	*12*	*0*	100%
polly/lib/Analysis	*9*	*9*	*0*	100%
polly/lib/CodeGen	*15*	*15*	*0*	100%
polly/lib/Exchange	*1*	*1*	*0*	100%
polly/lib/External/isl	*68*	*1*	*67*	1%
polly/lib/External/isl/imath	*6*	*1*	*5*	16%
polly/lib/External/isl/imath_wrap	*4*	*0*	*4*	0%
polly/lib/External/isl/include/isl	*59*	*9*	*50*	15%
polly/lib/External/isl/interface	*8*	*1*	*7*	12%
polly/lib/External/pet/include	*1*	*0*	*1*	0%
polly/lib/External/ppcg	*17*	*0*	*17*	0%
polly/lib/Plugin	*1*	*1*	*0*	100%
polly/lib/Support	*11*	*11*	*0*	100%
polly/lib/Transform	*15*	*15*	*0*	100%
polly/tools/GPURuntime	*1*	*1*	*0*	100%
polly/unittests/DeLICM	*1*	*1*	*0*	100%
polly/unittests/Flatten	*1*	*1*	*0*	100%
polly/unittests/Isl	*1*	*1*	*0*	100%
polly/unittests/ScheduleOptimizer	*1*	*1*	*0*	100%
polly/unittests/ScopPassManager	*1*	*1*	*0*	100%
polly/unittests/Support	*1*	*1*	*0*	100%
pstl/include/pstl/internal	*23*	*16*	*7*	69%
pstl/include/pstl/internal/omp	*11*	*8*	*3*	72%
third-party/benchmark/cmake	*5*	*1*	*4*	20%
third-party/benchmark/include/benchmark	*1*	*0*	*1*	0%
third-party/benchmark/src	*21*	*21*	*0*	100%
utils/bazel/llvm-project-overlay/clang/include/clang/Config	*1*	*1*	*0*	100%
utils/bazel/llvm-project-overlay/llvm/include/llvm/Config	*2*	*1*	*1*	50%
Total	16432	8857	7575	53%

Clang Linker Wrapper

Introduction 915

Usage 915

Example 915

Introduction

This tool works as a wrapper over a linking job. The tool is used to create linked device images for offloading. It scans the linker's input for embedded device offloading data stored in sections `.llvm.offloading.<triple>.<arch>` and extracts it as a temporary file. The extracted device files will then be passed to a device linking job to create a final device image. The sections will also be stripped and the resulting file passed back to the host linker.

Usage

This tool can be used with the following options. Arguments to the host linker being wrapper around are passed as positional arguments using the `--` flag to override parsing.

```
USAGE: clang-linker-wrapper [options] <options to be passed to linker>...

OPTIONS:

Generic Options:

  --help                    - Display available options (--help-hidden for more)
  --help-list               - Display list of available options (--help-list-hidden for more)
  --version                 - Display the version of this program

clang-linker-wrapper options:

  --host-triple=<string>     - Triple to use for the host compilation
  --linker-path=<string>     - Path of linker binary
  --opt-level=<string>       - Optimization level for LTO
  --ptxas-option=<string>    - Argument to pass to the ptxas invocation
  --save-temps               - Save intermediary results.
  --strip-sections           - Strip offloading sections from the host object file.
  --target-embed-bc          - Embed linked bitcode instead of an executable device image
  --target-feature=<string>  - Target features for triple
  --bitcode-library=<string> - Path for the target bitcode library
  -v                         - Verbose output from tools
```

Example

This tool links object files with offloading images embedded within it using the `-fembed-offload-object` flag in Clang. Given an input file containing the magic section we can pass it to this tool to extract the data contained at that section and run a device linking job on it.

```
clang-linker-wrapper -host-triple x86_64 -linker-path /usr/bin/ld -- <Args>
```

Clang Nvlink Wrapper

Introduction 915

Use Case 916

Working 916

Introduction

This tool works as a wrapper over the `nvlink` program. It is required because `nvlink` does not support linking of archive files implicitly. It transparently passes every input option and object to `nvlink` except archive files. It reads each input archive file to extract the archived cubin files as temporary files. These temporary (*.cubin) files are passed to `nvlink`.

Use Case

During linking of heterogeneous device archive libraries with an OpenMP program, the Clang Offload Bundler creates a device specific archive of cubin files. Such an archive is then passed to this wrapper tool to extract cubin files before passing to `nvlink`.

Working

Inputs

A command line generated by the OpenMP-Clang driver targeting NVPTX, containing a set of flags, cubin object files, and zero or more archive files.

Example:

```
clang-nvlink-wrapper main.cubin /tmp/libTest-nvptx-sm_50.a -o main-linked.out
```

Processing

1. From each archive file extract all cubin files as temporary files and store their names in a list, *CubinFiles*.
2. Create a new command line, *NVLinkCommand*, such that * Program is `nvlink` * All input flags are transparently passed on as flags * All input archive file are replaced with *CubinFiles*
3. Execute NVLinkCommand

```
1. Extract (libTest-nvptx-sm_50.a) => /tmp/a.cubin /tmp/b.cubin
2. nvlink -o a.out-openmp-nvptx64 main.cubin /tmp/a.cubin /tmp/b.cubin
```

Output

Output file generated by `nvlink` which links all cubin files.

Clang Offload Bundler

Introduction 916
Supported File Formats 917
Bundled Text File Layout 917
Bundled Binary File Layout 917
Bundle Entry ID 918
Target ID 919
Target Specific information 919
Archive Unbundling 920

Introduction

For heterogeneous single source programming languages, use one or more `--offload-arch=<target-id>` Clang options to specify the target IDs of the code to generate for the offload code regions.

The tool chain may perform multiple compilations of a translation unit to produce separate code objects for the host and potentially multiple offloaded devices. The `clang-offload-bundler` tool may be used as part of the tool chain to combine these multiple code objects into a single bundled code object.

The tool chain may use a bundled code object as an intermediate step so that each tool chain step consumes and produces a single file as in traditional non-heterogeneous tool chains. The bundled code object contains the code objects for the host and all the offload devices.

A bundled code object may also be used to bundle just the offloaded code objects, and embedded as data into the host code object. The host compilation includes an `init` function that will use the runtime corresponding to the offload kind (see Bundled Code Object Offload Kind) to load the offload code objects appropriate to the devices present when the host program is executed.

Supported File Formats

Several text and binary file formats are supported for bundling/unbundling. See Supported File Formats for a list of currently supported formats.

Supported File Formats

File Format	File Extension	Text/Binary
CPP output	i	Text
C++ CPP output	ii	Text
CUDA/HIP output	cui	Text
Dependency	d	Text
LLVM	ll	Text
LLVM Bitcode	bc	Binary
Assembler	s	Text
Object	o	Binary
Archive of objects	a	Binary
Precompiled header	gch	Binary
Clang AST file	ast	Binary

Bundled Text File Layout

The format of the bundled files is currently very simple: text formats are concatenated with comments that have a magic string and bundle entry ID in between.

```
"Comment OFFLOAD_BUNDLER_MAGIC_STR__START__ 1st Bundle Entry ID"
Bundle 1
"Comment OFFLOAD_BUNDLER_MAGIC_STR__END__ 1st Bundle Entry ID"
...
"Comment OFFLOAD_BUNDLER_MAGIC_STR__START__ Nth Bundle Entry ID"
Bundle N
"Comment OFFLOAD_BUNDLER_MAGIC_STR__END__ 1st Bundle Entry ID"
```

Bundled Binary File Layout

The layout of a bundled code object is defined by the following table:

Bundled Code Object Layout

Field	Type	Size in Bytes	Description
Magic String	string	24	`__CLANG_OFFLOAD_BUNDLE__`
Number Of Bundle Entries	integer	8	Number of bundle entries.
1st Bundle Entry Code Object Offset	integer	8	Byte offset from beginning of bundled code object to 1st code object.
1st Bundle Entry Code Object Size	integer	8	Byte size of 1st code object.
1st Bundle Entry ID Length	integer	8	Character length of bundle entry ID of 1st code object.

Field	Type	Size in Bytes	Description
1st Bundle Entry ID	string	1st Bundle Entry ID Length	Bundle entry ID of 1st code object. This is not NUL terminated. See Bundle Entry ID.
...			
Nth Bundle Entry Code Object Offset	integer	8	
Nth Bundle Entry Code Object Size	integer	8	
Nth Bundle Entry ID Length	integer	8	
Nth Bundle Entry ID	string	1st Bundle Entry ID Length	
1st Bundle Entry Code Object	bytes	1st Bundle Entry Code Object Size	
...			
Nth Bundle Entry Code Object	bytes	Nth Bundle Entry Code Object Size	

Bundle Entry ID

Each entry in a bundled code object (see Bundled Binary File Layout) has a bundle entry ID that indicates the kind of the entry's code object and the runtime that manages it.

Bundle entry ID syntax is defined by the following BNF syntax:

```
<bundle-entry-id> ::== <offload-kind> "-" <target-triple> [ "-" <target-id> ]
```

Where:

offload-kind

The runtime responsible for managing the bundled entry code object. See Bundled Code Object Offload Kind.

Bundled Code Object Offload Kind

Offload Kind	Description
host	Host code object. `clang-offload-bundler` always includes this entry as the first bundled code object entry. For an embedded bundled code object this entry is not used by the runtime and so is generally an empty code object.
hip	Offload code object for the HIP language. Used for all HIP language offload code objects when the `clang-offload-bundler` is used to bundle code objects as intermediate steps of the tool chain. Also used for AMD GPU code objects before ABI version V4 when the `clang-offload-bundler` is used to create a *fat binary* to be loaded by the HIP runtime. The fat binary can be loaded directly from a file, or be embedded in the host code object as a data section with the name `.hip_fatbin`.
hipv4	Offload code object for the HIP language. Used for AMD GPU code objects with at least ABI version V4 when the `clang-offload-bundler` is used to create a *fat binary* to be loaded by the HIP runtime. The fat binary can be loaded directly from a file, or be embedded in the host code object as a data section with the name `.hip_fatbin`.
openmp	Offload code object for the OpenMP language extension.

target-triple

The target triple of the code object.

target-id

The canonical target ID of the code object. Present only if the target supports a target ID. See Target ID.

Each entry of a bundled code object must have a different bundle entry ID. There can be multiple entries for the same processor provided they differ in target feature settings. If there is an entry with a target feature specified as *Any*, then all entries must specify that target feature as *Any* for the same processor. There may be additional target specific restrictions.

Target ID

A target ID is used to indicate the processor and optionally its configuration, expressed by a set of target features, that affect ISA generation. It is target specific if a target ID is supported, or if the target triple alone is sufficient to specify the ISA generation.

It is used with the `-mcpu=<target-id>` and `--offload-arch=<target-id>` Clang compilation options to specify the kind of code to generate.

It is also used as part of the bundle entry ID to identify the code object. See Bundle Entry ID.

Target ID syntax is defined by the following BNF syntax:

```
<target-id> ::== <processor> ( ":" <target-feature> ( "+" | "-" ) )*
```

Where:

processor

Is a the target specific processor or any alternative processor name.

target-feature

Is a target feature name that is supported by the processor. Each target feature must appear at most once in a target ID and can have one of three values:

Any

Specified by omitting the target feature from the target ID. A code object compiled with a target ID specifying the default value of a target feature can be loaded and executed on a processor configured with the target feature on or off.

On

Specified by +, indicating the target feature is enabled. A code object compiled with a target ID specifying a target feature on can only be loaded on a processor configured with the target feature on.

Off

specified by `-`, indicating the target feature is disabled. A code object compiled with a target ID specifying a target feature off can only be loaded on a processor configured with the target feature off.

There are two forms of target ID:

Non-Canonical Form

The non-canonical form is used as the input to user commands to allow the user greater convenience. It allows both the primary and alternative processor name to be used and the target features may be specified in any order.

Canonical Form

The canonical form is used for all generated output to allow greater convenience for tools that consume the information. It is also used for internal passing of information between tools. Only the primary and not alternative processor name is used and the target features are specified in alphabetic order. Command line tools convert non-canonical form to canonical form.

Target Specific information

Target specific information is available for the following:

AMD GPU

AMD GPU supports target ID and target features. See User Guide for AMDGPU Backend which defines the processors and target features supported.

Most other targets do not support target IDs.

Archive Unbundling

Unbundling of heterogeneous device archive is done to create device specific archives. Heterogeneous Device Archive is in a format compatible with GNU ar utility and contains a collection of bundled device binaries where each bundle file will contain device binaries for a host and one or more targets. The output device specific archive is in a format compatible with GNU ar utility and contains a collection of device binaries for a specific target.

```
Heterogeneous Device Archive, HDA = {F1.X, F2.X, ..., FN.Y}
where, Fi = Bundle{Host-DeviceBinary, T1-DeviceBinary, T2-DeviceBinary, ...,
                   Tm-DeviceBinary},
       Ti = {Target i, qualified using Bundle Entry ID},
       X/Y = \*.bc for AMDGPU and \*.cubin for NVPTX

Device Specific Archive, DSA(Tk) = {F1-Tk-DeviceBinary.X, F2-Tk-DeviceBinary.X, ...
                                    FN-Tk-DeviceBinary.Y}
where, Fi-Tj-DeviceBinary.X represents device binary of i-th bundled device
binary file for target Tj.
```

clang-offload-bundler extracts compatible device binaries for a given target from the bundled device binaries in a heterogeneous device archive and creates a target specific device archive without bundling.

clang-offload-bundler determines whether a device binary is compatible with a target by comparing bundle ID's. Two bundle ID's are considered compatible if:

- Their offload kind are the same
- Their target triple are the same
- Their GPUArch are the same

Clang Offload Wrapper

Introduction 920
Usage 920
Example 921
OpenMP Device Binary Embedding 921
Enum Types 921
Structure Types 921
Global Variables 922
Binary Descriptor for Device Images 922
Global Constructor and Destructor 923
Image Binary Embedding and Execution for OpenMP 923

Introduction

This tool is used in OpenMP offloading toolchain to embed device code objects (usually ELF) into a wrapper host llvm IR (bitcode) file. The wrapper host IR is then assembled and linked with host code objects to generate the executable binary. See Image Binary Embedding and Execution for OpenMP for more details.

Usage

This tool can be used as follows:

```
$ clang-offload-wrapper -help
OVERVIEW: A tool to create a wrapper bitcode for offload target binaries.
Takes offload target binaries as input and produces bitcode file containing
target binaries packaged as data and initialization code which registers
target binaries in offload runtime.
USAGE: clang-offload-wrapper [options] <input files>
```

```
OPTIONS:
Generic Options:
  --help                          - Display available options (--help-hidden for more)
  --help-list                     - Display list of available options (--help-list-hidden for more)
  --version                       - Display the version of this program
clang-offload-wrapper options:
  -o=<filename>                   - Output filename
  --target=<triple>               - Target triple for the output module
```

Example

```
clang-offload-wrapper -target host-triple -o host-wrapper.bc gfx90a-binary.out
```

OpenMP Device Binary Embedding

Various structures and functions used in the wrapper host IR form the interface between the executable binary and the OpenMP runtime.

Enum Types

Offloading Declare Target Flags Enum lists different flag for offloading entries.

Offloading Declare Target Flags Enum

Name	Value	Description
OMP_DECLARE_TARGET_LINK	0x01	Mark the entry as having a ‘link’ attribute (w.r.t. link clause)
OMP_DECLARE_TARGET_CTOR	0x02	Mark the entry as being a global constructor
OMP_DECLARE_TARGET_DTOR	0x04	Mark the entry as being a global destructor

Structure Types

__tgt_offload_entry structure, __tgt_device_image structure, and __tgt_bin_desc structure are the structures used in the wrapper host IR.

__tgt_offload_entry structure

Type	Identifier	Description
void*	addr	Address of global symbol within device image (function or global)
char*	name	Name of the symbol
size_t	size	Size of the entry info (0 if it is a function)
int32_t	flags	Flags associated with the entry (see Offloading Declare Target Flags Enum)
int32_t	reserved	Reserved, to be used by the runtime library.

__tgt_device_image structure

Type	Identifier	Description
void*	ImageStart	Pointer to the target code start

Type	Identifier	Description
void*	ImageEnd	Pointer to the target code end
__tgt_offload_entry*	EntriesBegin	Begin of table with all target entries
__tgt_offload_entry*	EntriesEnd	End of table (non inclusive)

__tgt_bin_desc structure

Type	Identifier	Description
int32_t	NumDeviceImages	Number of device types supported
__tgt_device_image*	DeviceImages	Array of device images (1 per dev. type)
__tgt_offload_entry*	HostEntriesBegin	Begin of table with all host entries
__tgt_offload_entry*	HostEntriesEnd	End of table (non inclusive)

Global Variables

Global Variables lists various global variables, along with their type and their explicit ELF sections, which are used to store device images and related symbols.

Global Variables

Variable	Type	ELF Section	Description
__start_omp_offloading_entries	__tgt_offload_entry	.omp_offloading_entries	Begin symbol for the offload entries table.
__stop_omp_offloading_entries	__tgt_offload_entry	.omp_offloading_entries	End symbol for the offload entries table.
__dummy.omp_offloading.entry	__tgt_offload_entry	.omp_offloading_entries	Dummy zero-sized object in the offload entries section to force linker to define begin/end symbols defined above.
.omp_offloading.device_image	__tgt_device_image	.omp_offloading_entries	ELF device code object of the first image.
.omp_offloading.device_image.N	__tgt_device_image	.omp_offloading_entries	ELF device code object of the (N+1)th image.
.omp_offloading.device_images	__tgt_device_image	.omp_offloading_entries	Array of images.
.omp_offloading.descriptor	__tgt_bin_desc	.omp_offloading_entries	Binary descriptor object (see details below).

Binary Descriptor for Device Images

This object is passed to the offloading runtime at program startup and it describes all device images available in the executable or shared library. It is defined as follows:

```
__attribute__((visibility("hidden")))
extern __tgt_offload_entry *__start_omp_offloading_entries;
__attribute__((visibility("hidden")))
extern __tgt_offload_entry *__stop_omp_offloading_entries;
static const char Image0[] = { <Bufs.front() contents> };
...
static const char ImageN[] = { <Bufs.back() contents> };
static const __tgt_device_image Images[] = {
  {
```

```
    Image0,                            /*ImageStart*/
    Image0 + sizeof(Image0),           /*ImageEnd*/
    __start_omp_offloading_entries,    /*EntriesBegin*/
    __stop_omp_offloading_entries      /*EntriesEnd*/
  },
  ...
  {
    ImageN,                            /*ImageStart*/
    ImageN + sizeof(ImageN),           /*ImageEnd*/
    __start_omp_offloading_entries,    /*EntriesBegin*/
    __stop_omp_offloading_entries      /*EntriesEnd*/
  }
};
static const __tgt_bin_desc BinDesc = {
  sizeof(Images) / sizeof(Images[0]),  /*NumDeviceImages*/
  Images,                              /*DeviceImages*/
  __start_omp_offloading_entries,      /*HostEntriesBegin*/
  __stop_omp_offloading_entries        /*HostEntriesEnd*/
};
```

Global Constructor and Destructor

Global constructor (`.omp_offloading.descriptor_reg()`) registers the library of images with the runtime by calling `__tgt_register_lib()` function. The cunstructor is explicitly defined in `.text.startup` section. Similarly, global destructor (`.omp_offloading.descriptor_unreg()`) calls `__tgt_unregister_lib()` for the unregistration and is also defined in `.text.startup` section.

Image Binary Embedding and Execution for OpenMP

For each offloading target, device ELF code objects are generated by `clang`, `opt`, `llc`, and `lld` pipeline. These code objects are passed to the `clang-offload-wrapper`.

- At compile time, the `clang-offload-wrapper` tool takes the following actions:
  - It embeds the ELF code objects for the device into the host code (see OpenMP Device Binary Embedding).
- At execution time:
  - The global constructor gets run and it registers the device image.

Clang Offload Packager

Introduction 923

Binary Format 924

Usage 925

Example 925

Introduction

This tool bundles device files into a single image containing necessary metadata. We use a custom binary format for bundling all the device images together. The image format is a small header wrapping around a string map. This tool creates bundled binaries so that they can be embedded into the host to create a fat-binary.

Binary Format

The binary format is marked by the `0x10FF10AD` magic bytes, followed by a version. Each created binary contains its own magic bytes. This allows us to locate all the embedded offloading sections even after they may have been merged by the linker, such as when using relocatable linking. Conceptually, this binary format is a serialization of a string map and an image buffer. The binary header is described in the following table.

Offloading Binary Header

Type	Identifier	Description
uint8_t	magic	The magic bytes for the binary format (0x10FF10AD)
uint32_t	version	Version of this format (currently version 1)
uint64_t	size	Size of this binary in bytes
uint64_t	entry offset	Absolute offset of the offload entries in bytes
uint64_t	entry size	Size of the offload entries in bytes

Once identified through the magic bytes, we use the size field to take a slice of the binary blob containing the information for a single offloading image. We can then use the offset field to find the actual offloading entries containing the image and metadata. The offload entry contains information about the device image. It contains the fields shown in the following table.

Offloading Entry Table

Type	Identifier	Description
uint16_t	image kind	The kind of the device image (e.g. bc, cubin)
uint16_t	offload kind	The producer of the image (e.g. openmp, cuda)
uint32_t	flags	Generic flags for the image
uint64_t	string offset	Absolute offset of the string metadata table
uint64_t	num strings	Number of string entries in the table
uint64_t	image offset	Absolute offset of the device image in bytes
uint64_t	image size	Size of the device image in bytes

This table contains the offsets of the string table and the device image itself along with some other integer information. The image kind lets us easily identify the type of image stored here without needing to inspect the binary. The offloading kind is used to determine which registration code or linking semantics are necessary for this image. These are stored as enumerations with the following values for the offload kind and the image kind.

Image Kind

Name	Value	Description
IMG_None	0x00	No image information provided
IMG_Object	0x01	The image is a generic object file
IMG_Bitcode	0x02	The image is an LLVM-IR bitcode file
IMG_Cubin	0x03	The image is a CUDA object file
IMG_Fatbinary	0x04	The image is a CUDA fatbinary file
IMG_PTX	0x05	The iamge is a CUDA PTX file

Offload Kind

Name	Value	Description
OFK_None	0x00	No offloading information provided
OFK_OpenMP	0x01	The producer was OpenMP offloading
OFK_CUDA	0x02	The producer was CUDA
OFK_HIP	0x03	The producer was HIP

The flags are used to signify certain conditions, such as the presence of debugging information or whether or not LTO was used. The string entry table is used to generically contain any arbitrary key-value pair. This is stored as an array of the string entry format.

Offloading String Entry

Type	Identifier	Description
uint64_t	key offset	Absolute byte offset of the key in th string table
uint64_t	value offset	Absolute byte offset of the value in the string table

The string entries simply provide offsets to a key and value pair in the binary images string table. The string table is simply a collection of null terminated strings with defined offsets in the image. The string entry allows us to create a key-value pair from this string table. This is used for passing arbitrary arguments to the image, such as the triple and architecture.

All of these structures are combined to form a single binary blob, the order does not matter because of the use of absolute offsets. This makes it easier to extend in the future. As mentioned previously, multiple offloading images are bundled together by simply concatenating them in this format. Because we have the magic bytes and size of each image, we can extract them as-needed.

Usage

This tool can be used with the following arguments. Generally information is passed as a key-value pair to the `image=` argument. The `file`, `triple`, and `arch` arguments are considered mandatory to make a valid image.

```
OVERVIEW: A utility for bundling several object files into a single binary.
The output binary can then be embedded into the host section table
to create a fatbinary containing offloading code.

USAGE: clang-offload-packager [options]

OPTIONS:

Generic Options:

  --help                      - Display available options (--help-hidden for more)
  --help-list                 - Display list of available options (--help-list-hidden for more)
  --version                   - Display the version of this program

clang-offload-packager options:

  --image=<<key>=<value>,...> - List of key and value arguments. Required
                                keywords are 'file' and 'triple'.
  -o=<file>                   - Write output to <file>.
```

Example

This tool simply takes many input files from the `image` option and creates a single output file with all the images combined.

```
clang-offload-packager -o out.bin --image=file=input.o,triple=nvptx64,arch=sm_70
```

Design Documents

“Clang” CFE Internals Manual

Introduction 927

LLVM Support Library 928

The Clang “Basic” Library 928

- The Diagnostics Subsystem 928
  - The `Diagnostic*Kinds.td` files 928
  - The Format String 929
  - Formatting a Diagnostic Argument 929
  - Producing the Diagnostic 932
  - Fix-It Hints 932
  - The `DiagnosticConsumer` Interface 933
  - Adding Translations to Clang 933
- The `SourceLocation` and `SourceManager` classes 933
- `SourceRange` and `CharSourceRange` 934

The Driver Library 934

Precompiled Headers 934

The Frontend Library 934

- Compiler Invocation 934
- Command Line Interface 934
- Command Line Parsing 935
- Command Line Generation 935
- Adding new Command Line Option 935
- Option Marshalling Infrastructure 937
- Option Marshalling Annotations 938

The Lexer and Preprocessor Library 939

- The Token class 940
- Annotation Tokens 940
- The `Lexer` class 941
- The `TokenLexer` class 942
- The `MultipleIncludeOpt` class 942

The Parser Library 942

The AST Library 942

- Design philosophy 942
  - Immutability 942
  - Faithfulness 943
- The `Type` class and its subclasses 943
  - Canonical Types 944
- The `QualType` class 944
- Declaration names 945
- Declaration contexts 946
  - Redeclarations and Overloads 946
  - Lexical and Semantic Contexts 947

Transparent Declaration Contexts 947
Multiply-Defined Declaration Contexts 948
Error Handling 949
Recovery AST 949
Types and dependence 950
ContainsErrors bit 950
The ASTImporter 951
Abstract Syntax Graph 951
Structural Equivalency 951
Redeclaration Chains 952
Traversal during the Import 953
Error Handling 953
Lookup Problems 954
ExternalASTSource 955
Class Template Instantiations 955
Visibility of Declarations 956
Strategies to Handle Conflicting Names 956
The `CFG` class 956
Basic Blocks 956
Entry and Exit Blocks 956
Conditional Control-Flow 957
Constant Folding in the Clang AST 958
Implementation Approach 958
Extensions 959
The Sema Library **959**
The CodeGen Library **959**
How to change Clang **959**
How to add an attribute 959
Attribute Basics 960
`include/clang/Basic/Attr.td` 960
Spellings 960
Subjects 961
Documentation 961
Arguments 962
Other Properties 962
Boilerplate 963
Semantic handling 963
How to add an expression or statement 963

Introduction

This document describes some of the more important APIs and internal design decisions made in the Clang C front-end. The purpose of this document is to both capture some of this high level information and also describe some of the design decisions behind it. This is meant for people interested in hacking on Clang, not for end-users. The description below is categorized by libraries, and does not describe any of the clients of the libraries.

LLVM Support Library

The LLVM `libSupport` library provides many underlying libraries and data-structures, including command line option processing, various containers and a system abstraction layer, which is used for file system access.

The Clang "Basic" Library

This library certainly needs a better name. The "basic" library contains a number of low-level utilities for tracking and manipulating source buffers, locations within the source buffers, diagnostics, tokens, target abstraction, and information about the subset of the language being compiled for.

Part of this infrastructure is specific to C (such as the `TargetInfo` class), other parts could be reused for other non-C-based languages (`SourceLocation`, `SourceManager`, `Diagnostics`, `FileManager`). When and if there is future demand we can figure out if it makes sense to introduce a new library, move the general classes somewhere else, or introduce some other solution.

We describe the roles of these classes in order of their dependencies.

The Diagnostics Subsystem

The Clang Diagnostics subsystem is an important part of how the compiler communicates with the human. Diagnostics are the warnings and errors produced when the code is incorrect or dubious. In Clang, each diagnostic produced has (at the minimum) a unique ID, an English translation associated with it, a SourceLocation to "put the caret", and a severity (e.g., `WARNING` or `ERROR`). They can also optionally include a number of arguments to the diagnostic (which fill in "%0"'s in the string) as well as a number of source ranges that related to the diagnostic.

In this section, we'll be giving examples produced by the Clang command line driver, but diagnostics can be rendered in many different ways depending on how the `DiagnosticConsumer` interface is implemented. A representative example of a diagnostic is:

```
t.c:38:15: error: invalid operands to binary expression ('int *' and '_Complex float')
P = (P-42) + Gamma*4;
    ~~~~~~ ^ ~~~~~~~
```

In this example, you can see the English translation, the severity (error), you can see the source location (the caret ("^") and file/line/column info), the source ranges "~~~~", arguments to the diagnostic ("`int*`" and "`_Complex float`"). You'll have to believe me that there is a unique ID backing the diagnostic :).

Getting all of this to happen has several steps and involves many moving pieces, this section describes them and talks about best practices when adding a new diagnostic.

The `Diagnostic*Kinds.td` files

Diagnostics are created by adding an entry to one of the `clang/Basic/Diagnostic*Kinds.td` files, depending on what library will be using it. From this file, **tblgen** generates the unique ID of the diagnostic, the severity of the diagnostic and the English translation + format string.

There is little sanity with the naming of the unique ID's right now. Some start with `err_`, `warn_`, `ext_` to encode the severity into the name. Since the enum is referenced in the C++ code that produces the diagnostic, it is somewhat useful for it to be reasonably short.

The severity of the diagnostic comes from the set {`NOTE`, `REMARK`, `WARNING`, `EXTENSION`, `EXTWARN`, `ERROR`}. The `ERROR` severity is used for diagnostics indicating the program is never acceptable under any circumstances. When an error is emitted, the AST for the input code may not be fully built. The `EXTENSION` and `EXTWARN` severities are used for extensions to the language that Clang accepts. This means that Clang fully understands and can represent them in the AST, but we produce diagnostics to tell the user their code is non-portable. The difference is that the former are ignored by default, and the later warn by default. The `WARNING` severity is used for constructs that are valid in the currently selected source language but that are dubious in some way. The `REMARK` severity provides generic information about the compilation that is not necessarily related to any dubious code. The `NOTE` level is used to staple more information onto previous diagnostics.

These *severities* are mapped into a smaller set (the `Diagnostic::Level` enum, {`Ignored`, `Note`, `Remark`, `Warning`, `Error`, `Fatal`}) of output *levels* by the diagnostics subsystem based on various configuration options. Clang internally supports a fully fine grained mapping mechanism that allows you to map almost any diagnostic to the output level that you want. The only diagnostics that cannot be mapped are `NOTE`s, which always follow the

severity of the previously emitted diagnostic and ERRORs, which can only be mapped to Fatal (it is not possible to turn an error into a warning, for example).

Diagnostic mappings are used in many ways. For example, if the user specifies -pedantic, EXTENSION maps to Warning, if they specify -pedantic-errors, it turns into Error. This is used to implement options like -Wunused_macros, -Wundef etc.

Mapping to Fatal should only be used for diagnostics that are considered so severe that error recovery won't be able to recover sensibly from them (thus spewing a ton of bogus errors). One example of this class of error are failure to #include a file.

The Format String

The format string for the diagnostic is very simple, but it has some power. It takes the form of a string in English with markers that indicate where and how arguments to the diagnostic are inserted and formatted. For example, here are some simple format strings:

```
"binary integer literals are an extension"
"format string contains '\\0' within the string body"
"more '%%' conversions than data arguments"
"invalid operands to binary expression (%0 and %1)"
"overloaded '%0' must be a %select{unary|binary|unary or binary}2 operator"
     " (has %1 parameter%s1)"
```

These examples show some important points of format strings. You can use any plain ASCII character in the diagnostic string except "%" without a problem, but these are C strings, so you have to use and be aware of all the C escape sequences (as in the second example). If you want to produce a "%" in the output, use the "%%" escape sequence, like the third diagnostic. Finally, Clang uses the "%...[digit]" sequences to specify where and how arguments to the diagnostic are formatted.

Arguments to the diagnostic are numbered according to how they are specified by the C++ code that produces them, and are referenced by %0 .. %9. If you have more than 10 arguments to your diagnostic, you are doing something wrong :). Unlike printf, there is no requirement that arguments to the diagnostic end up in the output in the same order as they are specified, you could have a format string with "%1 %0" that swaps them, for example. The text in between the percent and digit are formatting instructions. If there are no instructions, the argument is just turned into a string and substituted in.

Here are some "best practices" for writing the English format string:

- Keep the string short. It should ideally fit in the 80 column limit of the DiagnosticKinds.td file. This avoids the diagnostic wrapping when printed, and forces you to think about the important point you are conveying with the diagnostic.
- Take advantage of location information. The user will be able to see the line and location of the caret, so you don't need to tell them that the problem is with the 4th argument to the function: just point to it.
- Do not capitalize the diagnostic string, and do not end it with a period.
- If you need to quote something in the diagnostic string, use single quotes.

Diagnostics should never take random English strings as arguments: you shouldn't use "you have a problem with %0" and pass in things like "your argument" or "your return value" as arguments. Doing this prevents translating the Clang diagnostics to other languages (because they'll get random English words in their otherwise localized diagnostic). The exceptions to this are C/C++ language keywords (e.g., auto, const, mutable, etc) and C/C++ operators (/=). Note that things like "pointer" and "reference" are not keywords. On the other hand, you *can* include anything that comes from the user's source code, including variable names, types, labels, etc. The "select" format can be used to achieve this sort of thing in a localizable way, see below.

Formatting a Diagnostic Argument

Arguments to diagnostics are fully typed internally, and come from a couple different classes: integers, types, names, and random strings. Depending on the class of the argument, it can be optionally formatted in different ways. This gives the DiagnosticConsumer information about what the argument means without requiring it to use a specific presentation (consider this MVC for Clang :).

Here are the different diagnostic argument formats currently supported by Clang:

“s” format

Example:

```
"requires %1 parameter%s1"
```

Class:

Integers

Description:

This is a simple formatter for integers that is useful when producing English diagnostics. When the integer is 1, it prints as nothing. When the integer is not 1, it prints as “s”. This allows some simple grammatical forms to be to be handled correctly, and eliminates the need to use gross things like `"requires %1 parameter(s)"`.

“select” format

Example:

```
"must be a %select{unary|binary|unary or binary}2 operator"
```

Class:

Integers

Description:

This format specifier is used to merge multiple related diagnostics together into one common one, without requiring the difference to be specified as an English string argument. Instead of specifying the string, the diagnostic gets an integer argument and the format string selects the numbered option. In this case, the “%2” value must be an integer in the range [0..2]. If it is 0, it prints “unary”, if it is 1 it prints “binary” if it is 2, it prints “unary or binary”. This allows other language translations to substitute reasonable words (or entire phrases) based on the semantics of the diagnostic instead of having to do things textually. The selected string does undergo formatting.

“plural” format

Example:

```
"you have %1 %plural{1:mouse|:mice}1 connected to your computer"
```

Class:

Integers

Description:

This is a formatter for complex plural forms. It is designed to handle even the requirements of languages with very complex plural forms, as many Baltic languages have. The argument consists of a series of expression/form pairs, separated by “:”, where the first form whose expression evaluates to true is the result of the modifier.

An expression can be empty, in which case it is always true. See the example at the top. Otherwise, it is a series of one or more numeric conditions, separated by “,”. If any condition matches, the expression matches. Each numeric condition can take one of three forms.

- number: A simple decimal number matches if the argument is the same as the number. Example: `"%plural{1:mouse|:mice}4"`
- range: A range in square brackets matches if the argument is within the range. Then range is inclusive on both ends. Example: `"%plural{0:none|1:one|[2,5]:some|:many}2"`
- modulo: A modulo operator is followed by a number, and equals sign and either a number or a range. The tests are the same as for plain numbers and ranges, but the argument is taken modulo the number first. Example: `"%plural{%100=0:even hundred|%100=[1,50]:lower half|:everything else}1"`

The parser is very unforgiving. A syntax error, even whitespace, will abort, as will a failure to match the argument against any expression.

“ordinal” format

Example:

```
"ambiguity in %ordinal0 argument"
```

Class:

Integers

Description:

This is a formatter which represents the argument number as an ordinal: the value `1` becomes `1st`, `3` becomes `3rd`, and so on. Values less than `1` are not supported. This formatter is currently hard-coded to use English ordinals.

“objcclass” format

Example:

```
"method %objcclass0 not found"
```

Class:

`DeclarationName`

Description:

This is a simple formatter that indicates the `DeclarationName` corresponds to an Objective-C class method selector. As such, it prints the selector with a leading “+”.

“objcinstance” format

Example:

```
"method %objcinstance0 not found"
```

Class:

`DeclarationName`

Description:

This is a simple formatter that indicates the `DeclarationName` corresponds to an Objective-C instance method selector. As such, it prints the selector with a leading “`-`“.

“q” format

Example:

```
"candidate found by name lookup is %q0"
```

Class:

`NamedDecl *`

Description:

This formatter indicates that the fully-qualified name of the declaration should be printed, e.g., “`std::vector`” rather than “`vector`”.

“diff” format

Example:

```
"no known conversion %diff{from $ to $|from argument type to parameter type}1,2"
```

Class:

`QualType`

Description:

This formatter takes two `QualTypes` and attempts to print a template difference between the two. If tree printing is off, the text inside the braces before the pipe is printed, with the formatted text replacing the $. If tree printing is on, the text after the pipe is printed and a type tree is printed after the diagnostic message.

It is really easy to add format specifiers to the Clang diagnostics system, but they should be discussed before they are added. If you are creating a lot of repetitive diagnostics and/or have an idea for a useful formatter, please bring it up on the cfe-dev mailing list.

“sub” format

Example:

Given the following record definition of type `TextSubstitution`:

```
def select_ovl_candidate : TextSubstitution<
  "%select{function|constructor}0%select{| template| %2}1">;
```

which can be used as

```
def note_ovl_candidate : Note<
  "candidate %sub{select_ovl_candidate}3,2,1 not viable">;
```

and will act as if it was written `"candidate %select{function|constructor}3%select{| template| %1}2 not viable"`.

Description:

This format specifier is used to avoid repeating strings verbatim in multiple diagnostics. The argument to `%sub` must name a `TextSubstitution` tblgen record. The substitution must specify all arguments used by the substitution, and the modifier indexes in the substitution are re-numbered accordingly. The substituted text must itself be a valid format string before substitution.

Producing the Diagnostic

Now that you've created the diagnostic in the `Diagnostic*Kinds.td` file, you need to write the code that detects the condition in question and emits the new diagnostic. Various components of Clang (e.g., the preprocessor, `Sema`, etc.) provide a helper function named "`Diag`". It creates a diagnostic and accepts the arguments, ranges, and other information that goes along with it.

For example, the binary expression error comes from code like this:

```
if (various things that are bad)
  Diag(Loc, diag::err_typecheck_invalid_operands)
    << lex->getType() << rex->getType()
    << lex->getSourceRange() << rex->getSourceRange();
```

This shows that use of the `Diag` method: it takes a location (a SourceLocation object) and a diagnostic enum value (which matches the name from `Diagnostic*Kinds.td`). If the diagnostic takes arguments, they are specified with the << operator: the first argument becomes `%0`, the second becomes `%1`, etc. The diagnostic interface allows you to specify arguments of many different types, including `int` and `unsigned` for integer arguments, `const char*` and `std::string` for string arguments, `DeclarationName` and `const IdentifierInfo *` for names, `QualType` for types, etc. `SourceRanges` are also specified with the << operator, but do not have a specific ordering requirement.

As you can see, adding and producing a diagnostic is pretty straightforward. The hard part is deciding exactly what you need to say to help the user, picking a suitable wording, and providing the information needed to format it correctly. The good news is that the call site that issues a diagnostic should be completely independent of how the diagnostic is formatted and in what language it is rendered.

Fix-It Hints

In some cases, the front end emits diagnostics when it is clear that some small change to the source code would fix the problem. For example, a missing semicolon at the end of a statement or a use of deprecated syntax that is easily rewritten into a more modern form. Clang tries very hard to emit the diagnostic and recover gracefully in these and other cases.

However, for these cases where the fix is obvious, the diagnostic can be annotated with a hint (referred to as a "fix-it hint") that describes how to change the code referenced by the diagnostic to fix the problem. For example, it might add the missing semicolon at the end of the statement or rewrite the use of a deprecated construct into something more palatable. Here is one such example from the C++ front end, where we warn about the right-shift operator changing meaning from C++98 to C++11:

```
test.cpp:3:7: warning: use of right-shift operator ('>>') in template argument
                       will require parentheses in C++11
A<100 >> 2> *a;
      ^
  (       )
```

Here, the fix-it hint is suggesting that parentheses be added, and showing exactly where those parentheses would be inserted into the source code. The fix-it hints themselves describe what changes to make to the source code in an abstract manner, which the text diagnostic printer renders as a line of "insertions" below the caret line. Other diagnostic clients might choose to render the code differently (e.g., as markup inline) or even give the user the ability to automatically fix the problem.

Fix-it hints on errors and warnings need to obey these rules:

- Since they are automatically applied if `-Xclang -fixit` is passed to the driver, they should only be used when it's very likely they match the user's intent.

- Clang must recover from errors as if the fix-it had been applied.
- Fix-it hints on a warning must not change the meaning of the code. However, a hint may clarify the meaning as intentional, for example by adding parentheses when the precedence of operators isn't obvious.

If a fix-it can't obey these rules, put the fix-it on a note. Fix-its on notes are not applied automatically.

All fix-it hints are described by the `FixItHint` class, instances of which should be attached to the diagnostic using the << operator in the same way that highlighted source ranges and arguments are passed to the diagnostic. Fix-it hints can be created with one of three constructors:

- `FixItHint::CreateInsertion(Loc, Code)`

  Specifies that the given `Code` (a string) should be inserted before the source location `Loc`.

- `FixItHint::CreateRemoval(Range)`

  Specifies that the code in the given source `Range` should be removed.

- `FixItHint::CreateReplacement(Range, Code)`

  Specifies that the code in the given source `Range` should be removed, and replaced with the given `Code` string.

The `DiagnosticConsumer` Interface

Once code generates a diagnostic with all of the arguments and the rest of the relevant information, Clang needs to know what to do with it. As previously mentioned, the diagnostic machinery goes through some filtering to map a severity onto a diagnostic level, then (assuming the diagnostic is not mapped to "`Ignore`") it invokes an object that implements the `DiagnosticConsumer` interface with the information.

It is possible to implement this interface in many different ways. For example, the normal Clang `DiagnosticConsumer` (named `TextDiagnosticPrinter`) turns the arguments into strings (according to the various formatting rules), prints out the file/line/column information and the string, then prints out the line of code, the source ranges, and the caret. However, this behavior isn't required.

Another implementation of the `DiagnosticConsumer` interface is the `TextDiagnosticBuffer` class, which is used when Clang is in `-verify` mode. Instead of formatting and printing out the diagnostics, this implementation just captures and remembers the diagnostics as they fly by. Then `-verify` compares the list of produced diagnostics to the list of expected ones. If they disagree, it prints out its own output. Full documentation for the `-verify` mode can be found in the Clang API documentation for VerifyDiagnosticConsumer.

There are many other possible implementations of this interface, and this is why we prefer diagnostics to pass down rich structured information in arguments. For example, an HTML output might want declaration names be linkified to where they come from in the source. Another example is that a GUI might let you click on typedefs to expand them. This application would want to pass significantly more information about types through to the GUI than a simple flat string. The interface allows this to happen.

Adding Translations to Clang

Not possible yet! Diagnostic strings should be written in UTF-8, the client can translate to the relevant code page if needed. Each translation completely replaces the format string for the diagnostic.

The `SourceLocation` and `SourceManager` classes

Strangely enough, the `SourceLocation` class represents a location within the source code of the program. Important design points include:

1. `sizeof(SourceLocation)` must be extremely small, as these are embedded into many AST nodes and are passed around often. Currently it is 32 bits.
2. `SourceLocation` must be a simple value object that can be efficiently copied.
3. We should be able to represent a source location for any byte of any input file. This includes in the middle of tokens, in whitespace, in trigraphs, etc.
4. A `SourceLocation` must encode the current `#include` stack that was active when the location was processed. For example, if the location corresponds to a token, it should contain the set of `#includes` active when the token was lexed. This allows us to print the `#include` stack for a diagnostic.

5. `SourceLocation` must be able to describe macro expansions, capturing both the ultimate instantiation point and the source of the original character data.

In practice, the `SourceLocation` works together with the `SourceManager` class to encode two pieces of information about a location: its spelling location and its expansion location. For most tokens, these will be the same. However, for a macro expansion (or tokens that came from a `_Pragma` directive) these will describe the location of the characters corresponding to the token and the location where the token was used (i.e., the macro expansion point or the location of the `_Pragma` itself).

The Clang front-end inherently depends on the location of a token being tracked correctly. If it is ever incorrect, the front-end may get confused and die. The reason for this is that the notion of the "spelling" of a `Token` in Clang depends on being able to find the original input characters for the token. This concept maps directly to the "spelling location" for the token.

`SourceRange` and `CharSourceRange`

Clang represents most source ranges by [first, last], where "first" and "last" each point to the beginning of their respective tokens. For example consider the `SourceRange` of the following statement:

```
x = foo + bar;
^first    ^last
```

To map from this representation to a character-based representation, the "last" location needs to be adjusted to point to (or past) the end of that token with either `Lexer::MeasureTokenLength()` or `Lexer::getLocForEndOfToken()`. For the rare cases where character-level source ranges information is needed we use the `CharSourceRange` class.

The Driver Library

The clang Driver and library are documented here.

Precompiled Headers

Clang supports precompiled headers (PCH), which uses a serialized representation of Clang's internal data structures, encoded with the LLVM bitstream format.

The Frontend Library

The Frontend library contains functionality useful for building tools on top of the Clang libraries, for example several methods for outputting diagnostics.

Compiler Invocation

One of the classes provided by the Frontend library is `CompilerInvocation`, which holds information that describe current invocation of the Clang `-cc1` frontend. The information typically comes from the command line constructed by the Clang driver or from clients performing custom initialization. The data structure is split into logical units used by different parts of the compiler, for example `PreprocessorOptions`, `LanguageOptions` or `CodeGenOptions`.

Command Line Interface

The command line interface of the Clang `-cc1` frontend is defined alongside the driver options in `clang/Driver/Options.td`. The information making up an option definition includes its prefix and name (for example `-std=`), form and position of the option value, help text, aliases and more. Each option may belong to a certain group and can be marked with zero or more flags. Options accepted by the `-cc1` frontend are marked with the `CC1Option` flag.

Command Line Parsing

Option definitions are processed by the `-gen-opt-parser-defs` tablegen backend during early stages of the build. Options are then used for querying an instance `llvm::opt::ArgList`, a wrapper around the command line arguments. This is done in the Clang driver to construct individual jobs based on the driver arguments and also in the `CompilerInvocation::CreateFromArgs` function that parses the `-cc1` frontend arguments.

Command Line Generation

Any valid `CompilerInvocation` created from a `-cc1` command line can be also serialized back into semantically equivalent command line in a deterministic manner. This enables features such as implicitly discovered, explicitly built modules.

Adding new Command Line Option

When adding a new command line option, the first place of interest is the header file declaring the corresponding options class (e.g. `CodeGenOptions.h` for command line option that affects the code generation). Create new member variable for the option value:

```
  class CodeGenOptions : public CodeGenOptionsBase {

+   /// List of dynamic shared object files to be loaded as pass plugins.
+   std::vector<std::string> PassPlugins;

  }
```

Next, declare the command line interface of the option in the tablegen file `clang/include/clang/Driver/Options.td`. This is done by instantiating the `Option` class (defined in `llvm/include/llvm/Option/OptParser.td`). The instance is typically created through one of the helper classes that encode the acceptable ways to specify the option value on the command line:

- `Flag` - the option does not accept any value,
- `Joined` - the value must immediately follow the option name within the same argument,
- `Separate` - the value must follow the option name in the next command line argument,
- `JoinedOrSeparate` - the value can be specified either as `Joined` or `Separate`,
- `CommaJoined` - the values are comma-separated and must immediately follow the option name within the same argument (see `Wl,` for an example).

The helper classes take a list of acceptable prefixes of the option (e.g. `"-"`, `"--"` or `"/"`) and the option name:

```
  // Options.td

+ def fpass_plugin_EQ : Joined<["-"], "fpass-plugin=">;
```

Then, specify additional attributes via mix-ins:

- `HelpText` holds the text that will be printed besides the option name when the user requests help (e.g. via `clang --help`).
- `Group` specifies the “category” of options this option belongs to. This is used by various tools to filter certain options of interest.
- `Flags` may contain a number of “tags” associated with the option. This enables more granular filtering than the `Group` attribute.
- `Alias` denotes that the option is an alias of another option. This may be combined with `AliasArgs` that holds the implied value.

```
  // Options.td

  def fpass_plugin_EQ : Joined<["-"], "fpass-plugin=">,
+   Group<f_Group>, Flags<[CC1Option]>,
+   HelpText<"Load pass plugin from a dynamic shared object file.">;
```

New options are recognized by the Clang driver unless marked with the `NoDriverOption` flag. On the other hand, options intended for the `-cc1` frontend must be explicitly marked with the `CC1Option` flag.

Next, parse (or manufacture) the command line arguments in the Clang driver and use them to construct the `-cc1` job:

```
  void Clang::ConstructJob(const ArgList &Args /*...*/) const {
    ArgStringList CmdArgs;
    // ...

+   for (const Arg *A : Args.filtered(OPT_fpass_plugin_EQ)) {
+     CmdArgs.push_back(Args.MakeArgString(Twine("-fpass-plugin=") + A->getValue()));
+     A->claim();
+   }
  }
```

The last step is implementing the `-cc1` command line argument parsing/generation that initializes/serializes the option class (in our case `CodeGenOptions`) stored within `CompilerInvocation`. This can be done automatically by using the marshalling annotations on the option definition:

```
  // Options.td

  def fpass_plugin_EQ : Joined<["-"], "fpass-plugin=">,
    Group<f_Group>, Flags<[CC1Option]>,
    HelpText<"Load pass plugin from a dynamic shared object file.">,
+   MarshallingInfoStringVector<CodeGenOpts<"PassPlugins">>;
```

Inner workings of the system are introduced in the marshalling infrastructure section and the available annotations are listed here.

In case the marshalling infrastructure does not support the desired semantics, consider simplifying it to fit the existing model. This makes the command line more uniform and reduces the amount of custom, manually written code. Remember that the `-cc1` command line interface is intended only for Clang developers, meaning it does not need to mirror the driver interface, maintain backward compatibility or be compatible with GCC.

If the option semantics cannot be encoded via marshalling annotations, you can resort to parsing/serializing the command line arguments manually:

```
  // CompilerInvocation.cpp

  static bool ParseCodeGenArgs(CodeGenOptions &Opts, ArgList &Args /*...*/) {
    // ...

+   Opts.PassPlugins = Args.getAllArgValues(OPT_fpass_plugin_EQ);
  }

  static void GenerateCodeGenArgs(const CodeGenOptions &Opts,
                                  SmallVectorImpl<const char *> &Args,
                                  CompilerInvocation::StringAllocator SA /*...*/) {
    // ...

+   for (const std::string &PassPlugin : Opts.PassPlugins)
+     GenerateArg(Args, OPT_fpass_plugin_EQ, PassPlugin, SA);
  }
```

Finally, you can specify the argument on the command line: `clang -fpass-plugin=a -fpass-plugin=b` and use the new member variable as desired.

```
  void EmitAssemblyHelper::EmitAssemblyWithNewPassManager(/*...*/) {
    // ...
+   for (auto &PluginFN : CodeGenOpts.PassPlugins)
+     if (auto PassPlugin = PassPlugin::Load(PluginFN))
+        PassPlugin->registerPassBuilderCallbacks(PB);
  }
```

Option Marshalling Infrastructure

The option marshalling infrastructure automates the parsing of the Clang `-cc1` frontend command line arguments into `CompilerInvocation` and their generation from `CompilerInvocation`. The system replaces lots of repetitive C++ code with simple, declarative tablegen annotations and it's being used for the majority of the `-cc1` command line interface. This section provides an overview of the system.

Note: The marshalling infrastructure is not intended for driver-only options. Only options of the `-cc1` frontend need to be marshalled to/from `CompilerInvocation` instance.

To read and modify contents of `CompilerInvocation`, the marshalling system uses key paths, which are declared in two steps. First, a tablegen definition for the `CompilerInvocation` member is created by inheriting from `KeyPathAndMacro`:

```
// Options.td

class LangOpts<string field> : KeyPathAndMacro<"LangOpts->", field, "LANG_"> {}
//                   CompilerInvocation member  ^^^^^^^^^^
//                                     OPTION_WITH_MARSHALLING prefix ^^^^^
```

The first argument to the parent class is the beginning of the key path that references the `CompilerInvocation` member. This argument ends with `->` if the member is a pointer type or with `.` if it's a value type. The child class takes a single parameter `field` that is forwarded as the second argument to the base class. The child class can then be used like so: `LangOpts<"IgnoreExceptions">`, constructing a key path to the field `LangOpts->IgnoreExceptions`. The third argument passed to the parent class is a string that the tablegen backend uses as a prefix to the `OPTION_WITH_MARSHALLING` macro. Using the key path as a mix-in on an `Option` instance instructs the backend to generate the following code:

```
// Options.inc

#ifdef LANG_OPTION_WITH_MARSHALLING
LANG_OPTION_WITH_MARSHALLING([...], LangOpts->IgnoreExceptions, [...])
#endif // LANG_OPTION_WITH_MARSHALLING
```

Such definition can be used used in the function for parsing and generating command line:

```
// clang/lib/Frontend/CompilerInvoation.cpp

bool CompilerInvocation::ParseLangArgs(LangOptions *LangOpts, ArgList &Args,
                                       DiagnosticsEngine &Diags) {
  bool Success = true;

#define LANG_OPTION_WITH_MARSHALLING(                                          \
    PREFIX_TYPE, NAME, ID, KIND, GROUP, ALIAS, ALIASARGS, FLAGS, PARAM,        \
    HELPTEXT, METAVAR, VALUES, SPELLING, SHOULD_PARSE, ALWAYS_EMIT, KEYPATH,   \
    DEFAULT_VALUE, IMPLIED_CHECK, IMPLIED_VALUE, NORMALIZER, DENORMALIZER,     \
    MERGER, EXTRACTOR, TABLE_INDEX)                                            \
  PARSE_OPTION_WITH_MARSHALLING(Args, Diags, Success, ID, FLAGS, PARAM,        \
                                SHOULD_PARSE, KEYPATH, DEFAULT_VALUE,          \
                                IMPLIED_CHECK, IMPLIED_VALUE, NORMALIZER,      \
                                MERGER, TABLE_INDEX)
#include "clang/Driver/Options.inc"
#undef LANG_OPTION_WITH_MARSHALLING

  // ...

  return Success;
}

void CompilerInvocation::GenerateLangArgs(LangOptions *LangOpts,
                                          SmallVectorImpl<const char *> &Args,
                                          StringAllocator SA) {
#define LANG_OPTION_WITH_MARSHALLING(                                          \
    PREFIX_TYPE, NAME, ID, KIND, GROUP, ALIAS, ALIASARGS, FLAGS, PARAM,        \
```

```
    HELPTEXT, METAVAR, VALUES, SPELLING, SHOULD_PARSE, ALWAYS_EMIT, KEYPATH,    \
    DEFAULT_VALUE, IMPLIED_CHECK, IMPLIED_VALUE, NORMALIZER, DENORMALIZER,      \
    MERGER, EXTRACTOR, TABLE_INDEX)                                             \
  GENERATE_OPTION_WITH_MARSHALLING(                                             \
      Args, SA, KIND, FLAGS, SPELLING, ALWAYS_EMIT, KEYPATH, DEFAULT_VALUE,    \
      IMPLIED_CHECK, IMPLIED_VALUE, DENORMALIZER, EXTRACTOR, TABLE_INDEX)
#include "clang/Driver/Options.inc"
#undef LANG_OPTION_WITH_MARSHALLING

  // ...
}
```

The `PARSE_OPTION_WITH_MARSHALLING` and `GENERATE_OPTION_WITH_MARSHALLING` macros are defined in `CompilerInvocation.cpp` and they implement the generic algorithm for parsing and generating command line arguments.

Option Marshalling Annotations

How does the tablegen backend know what to put in place of `[...]` in the generated `Options.inc`? This is specified by the `Marshalling` utilities described below. All of them take a key path argument and possibly other information required for parsing or generating the command line argument.

Note: The marshalling infrastructure is not intended for driver-only options. Only options of the `-cc1` frontend need to be marshalled to/from `CompilerInvocation` instance.

Positive Flag

The key path defaults to `false` and is set to `true` when the flag is present on command line.

```
def fignore_exceptions : Flag<["-"], "fignore-exceptions">, Flags<[CC1Option]>,
  MarshallingInfoFlag<LangOpts<"IgnoreExceptions">>;
```

Negative Flag

The key path defaults to `true` and is set to `false` when the flag is present on command line.

```
def fno_verbose_asm : Flag<["-"], "fno-verbose-asm">, Flags<[CC1Option]>,
  MarshallingInfoNegativeFlag<CodeGenOpts<"AsmVerbose">>;
```

Negative and Positive Flag

The key path defaults to the specified value (`false`, `true` or some boolean value that's statically unknown in the tablegen file). Then, the key path is set to the value associated with the flag that appears last on command line.

```
defm legacy_pass_manager : BoolOption<"f", "legacy-pass-manager",
  CodeGenOpts<"LegacyPassManager">, DefaultFalse,
  PosFlag<SetTrue, [], "Use the legacy pass manager in LLVM">,
  NegFlag<SetFalse, [], "Use the new pass manager in LLVM">,
  BothFlags<[CC1Option]>>;
```

With most such pair of flags, the `-cc1` frontend accepts only the flag that changes the default key path value. The Clang driver is responsible for accepting both and either forwarding the changing flag or discarding the flag that would just set the key path to its default.

The first argument to `BoolOption` is a prefix that is used to construct the full names of both flags. The positive flag would then be named `flegacy-pass-manager` and the negative `fno-legacy-pass-manager`. `BoolOption` also implies the `-` prefix for both flags. It's also possible to use `BoolFOption` that implies the `"f"` prefix and `Group<f_Group>`. The `PosFlag` and `NegFlag` classes hold the associated boolean value, an array of elements passed to the `Flag` class and the help text. The optional `BothFlags` class holds an array of `Flag` elements that are common for both the positive and negative flag and their common help text suffix.

String

The key path defaults to the specified string, or an empty one, if omitted. When the option appears on the command line, the argument value is simply copied.

```
def isysroot : JoinedOrSeparate<["-"], "isysroot">, Flags<[CC1Option]>,
  MarshallingInfoString<HeaderSearchOpts<"Sysroot">, [{"/"}]>;
```

List of Strings

The key path defaults to an empty `std::vector<std::string>`. Values specified with each appearance of the option on the command line are appended to the vector.

```
def frewrite_map_file : Separate<["-"], "frewrite-map-file">, Flags<[CC1Option]>,
  MarshallingInfoStringVector<CodeGenOpts<"RewriteMapFiles">>;
```

Integer

The key path defaults to the specified integer value, or `0` if omitted. When the option appears on the command line, its value gets parsed by `llvm::APInt` and the result is assigned to the key path on success.

```
def mstack_probe_size : Joined<["-"], "mstack-probe-size=">, Flags<[CC1Option]>,
  MarshallingInfoInt<CodeGenOpts<"StackProbeSize">, "4096">;
```

Enumeration

The key path defaults to the value specified in `MarshallingInfoEnum` prefixed by the contents of `NormalizedValuesScope` and `::`. This ensures correct reference to an enum case is formed even if the enum resides in different namespace or is an enum class. If the value present on command line does not match any of the comma-separated values from `Values`, an error diagnostics is issued. Otherwise, the corresponding element from `NormalizedValues` at the same index is assigned to the key path (also correctly scoped). The number of comma-separated string values and elements of the array within `NormalizedValues` must match.

```
def mthread_model : Separate<["-"], "mthread-model">, Flags<[CC1Option]>,
  Values<"posix,single">, NormalizedValues<["POSIX", "Single"]>,
  NormalizedValuesScope<"LangOptions::ThreadModelKind">,
  MarshallingInfoEnum<LangOpts<"ThreadModel">, "POSIX">;
```

It is also possible to define relationships between options.

Implication

The key path defaults to the default value from the primary `Marshalling` annotation. Then, if any of the elements of `ImpliedByAnyOf` evaluate to true, the key path value is changed to the specified value or `true` if missing. Finally, the command line is parsed according to the primary annotation.

```
def fms_extensions : Flag<["-"], "fms-extensions">, Flags<[CC1Option]>,
  MarshallingInfoFlag<LangOpts<"MicrosoftExt">>,
  ImpliedByAnyOf<[fms_compatibility.KeyPath], "true">;
```

Condition

The option is parsed only if the expression in `ShouldParseIf` evaluates to true.

```
def fopenmp_enable_irbuilder : Flag<["-"], "fopenmp-enable-irbuilder">, Flags<[CC1Option]>,
  MarshallingInfoFlag<LangOpts<"OpenMPIRBuilder">>,
  ShouldParseIf<fopenmp.KeyPath>;
```

The Lexer and Preprocessor Library

The Lexer library contains several tightly-connected classes that are involved with the nasty process of lexing and preprocessing C source code. The main interface to this library for outside clients is the large `Preprocessor` class. It contains the various pieces of state that are required to coherently read tokens out of a translation unit.

The core interface to the `Preprocessor` object (once it is set up) is the `Preprocessor::Lex` method, which returns the next Token from the preprocessor stream. There are two types of token providers that the preprocessor is capable of reading from: a buffer lexer (provided by the Lexer class) and a buffered token stream (provided by the TokenLexer class).

The Token class

The `Token` class is used to represent a single lexed token. Tokens are intended to be used by the lexer/preprocess and parser libraries, but are not intended to live beyond them (for example, they should not live in the ASTs).

Tokens most often live on the stack (or some other location that is efficient to access) as the parser is running, but occasionally do get buffered up. For example, macro definitions are stored as a series of tokens, and the C++ front-end periodically needs to buffer tokens up for tentative parsing and various pieces of look-ahead. As such, the size of a `Token` matters. On a 32-bit system, `sizeof(Token)` is currently 16 bytes.

Tokens occur in two forms: annotation tokens and normal tokens. Normal tokens are those returned by the lexer, annotation tokens represent semantic information and are produced by the parser, replacing normal tokens in the token stream. Normal tokens contain the following information:

- **A SourceLocation** — This indicates the location of the start of the token.
- **A length** — This stores the length of the token as stored in the `SourceBuffer`. For tokens that include them, this length includes trigraphs and escaped newlines which are ignored by later phases of the compiler. By pointing into the original source buffer, it is always possible to get the original spelling of a token completely accurately.
- **IdentifierInfo** — If a token takes the form of an identifier, and if identifier lookup was enabled when the token was lexed (e.g., the lexer was not reading in "raw" mode) this contains a pointer to the unique hash value for the identifier. Because the lookup happens before keyword identification, this field is set even for language keywords like "`for`".
- **TokenKind** — This indicates the kind of token as classified by the lexer. This includes things like `tok::starequal` (for the "`*=`" operator), `tok::ampamp` for the "`&&`" token, and keyword values (e.g., `tok::kw_for`) for identifiers that correspond to keywords. Note that some tokens can be spelled multiple ways. For example, C++ supports "operator keywords", where things like "`and`" are treated exactly like the "`&&`" operator. In these cases, the kind value is set to `tok::ampamp`, which is good for the parser, which doesn't have to consider both forms. For something that cares about which form is used (e.g., the preprocessor "stringize" operator) the spelling indicates the original form.
- **Flags** — There are currently four flags tracked by the lexer/preprocessor system on a per-token basis:

  1. **StartOfLine** — This was the first token that occurred on its input source line.
  2. **LeadingSpace** — There was a space character either immediately before the token or transitively before the token as it was expanded through a macro. The definition of this flag is very closely defined by the stringizing requirements of the preprocessor.
  3. **DisableExpand** — This flag is used internally to the preprocessor to represent identifier tokens which have macro expansion disabled. This prevents them from being considered as candidates for macro expansion ever in the future.
  4. **NeedsCleaning** — This flag is set if the original spelling for the token includes a trigraph or escaped newline. Since this is uncommon, many pieces of code can fast-path on tokens that did not need cleaning.

One interesting (and somewhat unusual) aspect of normal tokens is that they don't contain any semantic information about the lexed value. For example, if the token was a pp-number token, we do not represent the value of the number that was lexed (this is left for later pieces of code to decide). Additionally, the lexer library has no notion of typedef names vs variable names: both are returned as identifiers, and the parser is left to decide whether a specific identifier is a typedef or a variable (tracking this requires scope information among other things). The parser can do this translation by replacing tokens returned by the preprocessor with "Annotation Tokens".

Annotation Tokens

Annotation tokens are tokens that are synthesized by the parser and injected into the preprocessor's token stream (replacing existing tokens) to record semantic information found by the parser. For example, if "`foo`" is found to be a typedef, the "`foo`" `tok::identifier` token is replaced with an `tok::annot_typename`. This is useful for a couple of reasons: 1) this makes it easy to handle qualified type names (e.g., "`foo::bar::baz<42>::t`") in C++ as a single "token" in the parser. 2) if the parser backtracks, the reparse does not need to redo semantic analysis to determine whether a token sequence is a variable, type, template, etc.

Annotation tokens are created by the parser and reinjected into the parser's token stream (when backtracking is enabled). Because they can only exist in tokens that the preprocessor-proper is done with, it doesn't need to keep

around flags like “start of line” that the preprocessor uses to do its job. Additionally, an annotation token may “cover” a sequence of preprocessor tokens (e.g., “`a::b::c`” is five preprocessor tokens). As such, the valid fields of an annotation token are different than the fields for a normal token (but they are multiplexed into the normal `Token` fields):

- **SourceLocation “Location”** — The `SourceLocation` for the annotation token indicates the first token replaced by the annotation token. In the example above, it would be the location of the “a” identifier.
- **SourceLocation “AnnotationEndLoc”** — This holds the location of the last token replaced with the annotation token. In the example above, it would be the location of the “c” identifier.
- **void* “AnnotationValue”** — This contains an opaque object that the parser gets from `Sema`. The parser merely preserves the information for `Sema` to later interpret based on the annotation token kind.
- **TokenKind “Kind”** — This indicates the kind of Annotation token this is. See below for the different valid kinds.

Annotation tokens currently come in three kinds:

1. **tok::annot_typename**: This annotation token represents a resolved typename token that is potentially qualified. The `AnnotationValue` field contains the `QualType` returned by `Sema::getTypeName()`, possibly with source location information attached.
2. **tok::annot_cxxscope**: This annotation token represents a C++ scope specifier, such as “`A::B::`”. This corresponds to the grammar productions “*::*” and “*:: [opt] nested-name-specifier*”. The `AnnotationValue` pointer is a `NestedNameSpecifier *` returned by the `Sema::ActOnCXXGlobalScopeSpecifier` and `Sema::ActOnCXXNestedNameSpecifier` callbacks.
3. **tok::annot_template_id**: This annotation token represents a C++ template-id such as “`foo<int, 4>`”, where “`foo`” is the name of a template. The `AnnotationValue` pointer is a pointer to a `malloc`’d `TemplateIdAnnotation` object. Depending on the context, a parsed template-id that names a type might become a typename annotation token (if all we care about is the named type, e.g., because it occurs in a type specifier) or might remain a template-id token (if we want to retain more source location information or produce a new type, e.g., in a declaration of a class template specialization). template-id annotation tokens that refer to a type can be “upgraded” to typename annotation tokens by the parser.

As mentioned above, annotation tokens are not returned by the preprocessor, they are formed on demand by the parser. This means that the parser has to be aware of cases where an annotation could occur and form it where appropriate. This is somewhat similar to how the parser handles Translation Phase 6 of C99: String Concatenation (see C99 5.1.1.2). In the case of string concatenation, the preprocessor just returns distinct `tok::string_literal` and `tok::wide_string_literal` tokens and the parser eats a sequence of them wherever the grammar indicates that a string literal can occur.

In order to do this, whenever the parser expects a `tok::identifier` or `tok::coloncolon`, it should call the `TryAnnotateTypeOrScopeToken` or `TryAnnotateCXXScopeToken` methods to form the annotation token. These methods will maximally form the specified annotation tokens and replace the current token with them, if applicable. If the current tokens is not valid for an annotation token, it will remain an identifier or “`::`” token.

The `Lexer` class

The `Lexer` class provides the mechanics of lexing tokens out of a source buffer and deciding what they mean. The `Lexer` is complicated by the fact that it operates on raw buffers that have not had spelling eliminated (this is a necessity to get decent performance), but this is countered with careful coding as well as standard performance techniques (for example, the comment handling code is vectorized on X86 and PowerPC hosts).

The lexer has a couple of interesting modal features:

- The lexer can operate in “raw” mode. This mode has several features that make it possible to quickly lex the file (e.g., it stops identifier lookup, doesn’t specially handle preprocessor tokens, handles EOF differently, etc). This mode is used for lexing within an “`#if 0`” block, for example.
- The lexer can capture and return comments as tokens. This is required to support the `-C` preprocessor mode, which passes comments through, and is used by the diagnostic checker to identifier expect-error annotations.
- The lexer can be in `ParsingFilename` mode, which happens when preprocessing after reading a `#include` directive. This mode changes the parsing of “<” to return an “angled string” instead of a bunch of tokens for each thing within the filename.

- When parsing a preprocessor directive (after "#") the `ParsingPreprocessorDirective` mode is entered. This changes the parser to return EOD at a newline.
- The `Lexer` uses a `LangOptions` object to know whether trigraphs are enabled, whether C++ or ObjC keywords are recognized, etc.

In addition to these modes, the lexer keeps track of a couple of other features that are local to a lexed buffer, which change as the buffer is lexed:

- The `Lexer` uses `BufferPtr` to keep track of the current character being lexed.
- The `Lexer` uses `IsAtStartOfLine` to keep track of whether the next lexed token will start with its "start of line" bit set.
- The `Lexer` keeps track of the current "`#if`" directives that are active (which can be nested).
- The `Lexer` keeps track of an MultipleIncludeOpt object, which is used to detect whether the buffer uses the standard "`#ifndef XX` / `#define XX`" idiom to prevent multiple inclusion. If a buffer does, subsequent includes can be ignored if the "XX" macro is defined.

The `TokenLexer` class

The `TokenLexer` class is a token provider that returns tokens from a list of tokens that came from somewhere else. It typically used for two things: 1) returning tokens from a macro definition as it is being expanded 2) returning tokens from an arbitrary buffer of tokens. The later use is used by `_Pragma` and will most likely be used to handle unbounded look-ahead for the C++ parser.

The `MultipleIncludeOpt` class

The `MultipleIncludeOpt` class implements a really simple little state machine that is used to detect the standard "`#ifndef XX` / `#define XX`" idiom that people typically use to prevent multiple inclusion of headers. If a buffer uses this idiom and is subsequently `#include`'d, the preprocessor can simply check to see whether the guarding condition is defined or not. If so, the preprocessor can completely ignore the include of the header.

The Parser Library

This library contains a recursive-descent parser that polls tokens from the preprocessor and notifies a client of the parsing progress.

Historically, the parser used to talk to an abstract `Action` interface that had virtual methods for parse events, for example `ActOnBinOp()`. When Clang grew C++ support, the parser stopped supporting general `Action` clients – it now always talks to the Sema library. However, the Parser still accesses AST objects only through opaque types like `ExprResult` and `StmtResult`. Only Sema looks at the AST node contents of these wrappers.

The AST Library

Design philosophy

Immutability

Clang AST nodes (types, declarations, statements, expressions, and so on) are generally designed to be immutable once created. This provides a number of key benefits:

- Canonicalization of the "meaning" of nodes is possible as soon as the nodes are created, and is not invalidated by later addition of more information. For example, we canonicalize types, and use a canonicalized representation of expressions when determining whether two function template declarations involving dependent expressions declare the same entity.
- AST nodes can be reused when they have the same meaning. For example, we reuse `Type` nodes when representing the same type (but maintain separate `TypeLocs` for each instance where a type is written), and we reuse non-dependent `Stmt` and `Expr` nodes across instantiations of a template.
- Serialization and deserialization of the AST to/from AST files is simpler: we do not need to track modifications made to AST nodes imported from AST files and serialize separate "update records".

There are unfortunately exceptions to this general approach, such as:

- The first declaration of a redeclarable entity maintains a pointer to the most recent declaration of that entity, which naturally needs to change as more declarations are parsed.
- Name lookup tables in declaration contexts change after the namespace declaration is formed.
- We attempt to maintain only a single declaration for an instantiation of a template, rather than having distinct declarations for an instantiation of the declaration versus the definition, so template instantiation often updates parts of existing declarations.
- Some parts of declarations are required to be instantiated separately (this includes default arguments and exception specifications), and such instantiations update the existing declaration.

These cases tend to be fragile; mutable AST state should be avoided where possible.

As a consequence of this design principle, we typically do not provide setters for AST state. (Some are provided for short-term modifications intended to be used immediately after an AST node is created and before it's "published" as part of the complete AST, or where language semantics require after-the-fact updates.)

Faithfulness

The AST intends to provide a representation of the program that is faithful to the original source. We intend for it to be possible to write refactoring tools using only information stored in, or easily reconstructible from, the Clang AST. This means that the AST representation should either not desugar source-level constructs to simpler forms, or – where made necessary by language semantics or a clear engineering tradeoff – should desugar minimally and wrap the result in a construct representing the original source form.

For example, `CXXForRangeStmt` directly represents the syntactic form of a range-based for statement, but also holds a semantic representation of the range declaration and iterator declarations. It does not contain a fully-desugared `ForStmt`, however.

Some AST nodes (for example, `ParenExpr`) represent only syntax, and others (for example, `ImplicitCastExpr`) represent only semantics, but most nodes will represent a combination of syntax and associated semantics. Inheritance is typically used when representing different (but related) syntaxes for nodes with the same or similar semantics.

The `Type` class and its subclasses

The `Type` class (and its subclasses) are an important part of the AST. Types are accessed through the `ASTContext` class, which implicitly creates and uniques them as they are needed. Types have a couple of non-obvious features: 1) they do not capture type qualifiers like `const` or `volatile` (see QualType), and 2) they implicitly capture typedef information. Once created, types are immutable (unlike decls).

Typedefs in C make semantic analysis a bit more complex than it would be without them. The issue is that we want to capture typedef information and represent it in the AST perfectly, but the semantics of operations need to "see through" typedefs. For example, consider this code:

```
void func() {
  typedef int foo;
  foo X, *Y;
  typedef foo *bar;
  bar Z;
  *X; // error
  **Y; // error
  **Z; // error
}
```

The code above is illegal, and thus we expect there to be diagnostics emitted on the annotated lines. In this example, we expect to get:

```
test.c:6:1: error: indirection requires pointer operand ('foo' invalid)
  *X; // error
  ^~
test.c:7:1: error: indirection requires pointer operand ('foo' invalid)
  **Y; // error
```

```
  ^~~
test.c:8:1: error: indirection requires pointer operand ('foo' invalid)
  **Z; // error
  ^~~
```

While this example is somewhat silly, it illustrates the point: we want to retain typedef information where possible, so that we can emit errors about "`std::string`" instead of "`std::basic_string<char, std:...`". Doing this requires properly keeping typedef information (for example, the type of `X` is "`foo`", not "`int`"), and requires properly propagating it through the various operators (for example, the type of `*Y` is "`foo`", not "`int`"). In order to retain this information, the type of these expressions is an instance of the `TypedefType` class, which indicates that the type of these expressions is a typedef for "`foo`".

Representing types like this is great for diagnostics, because the user-specified type is always immediately available. There are two problems with this: first, various semantic checks need to make judgements about the *actual structure* of a type, ignoring typedefs. Second, we need an efficient way to query whether two types are structurally identical to each other, ignoring typedefs. The solution to both of these problems is the idea of canonical types.

Canonical Types

Every instance of the `Type` class contains a canonical type pointer. For simple types with no typedefs involved (e.g., "`int`", "`int*`", "`int**`"), the type just points to itself. For types that have a typedef somewhere in their structure (e.g., "`foo`", "`foo*`", "`foo**`", "`bar`"), the canonical type pointer points to their structurally equivalent type without any typedefs (e.g., "`int`", "`int*`", "`int**`", and "`int*`" respectively).

This design provides a constant time operation (dereferencing the canonical type pointer) that gives us access to the structure of types. For example, we can trivially tell that "`bar`" and "`foo*`" are the same type by dereferencing their canonical type pointers and doing a pointer comparison (they both point to the single "`int*`" type).

Canonical types and typedef types bring up some complexities that must be carefully managed. Specifically, the `isa`/`cast`/`dyn_cast` operators generally shouldn't be used in code that is inspecting the AST. For example, when type checking the indirection operator (unary "`*`" on a pointer), the type checker must verify that the operand has a pointer type. It would not be correct to check that with "`isa<PointerType>(SubExpr->getType())`", because this predicate would fail if the subexpression had a typedef type.

The solution to this problem are a set of helper methods on `Type`, used to check their properties. In this case, it would be correct to use "`SubExpr->getType()->isPointerType()`" to do the check. This predicate will return true if the *canonical type is a pointer*, which is true any time the type is structurally a pointer type. The only hard part here is remembering not to use the `isa`/`cast`/`dyn_cast` operations.

The second problem we face is how to get access to the pointer type once we know it exists. To continue the example, the result type of the indirection operator is the pointee type of the subexpression. In order to determine the type, we need to get the instance of `PointerType` that best captures the typedef information in the program. If the type of the expression is literally a `PointerType`, we can return that, otherwise we have to dig through the typedefs to find the pointer type. For example, if the subexpression had type "`foo*`", we could return that type as the result. If the subexpression had type "`bar`", we want to return "`foo*`" (note that we do *not* want "`int*`"). In order to provide all of this, `Type` has a `getAsPointerType()` method that checks whether the type is structurally a `PointerType` and, if so, returns the best one. If not, it returns a null pointer.

This structure is somewhat mystical, but after meditating on it, it will make sense to you :).

The `QualType` class

The `QualType` class is designed as a trivial value class that is small, passed by-value and is efficient to query. The idea of `QualType` is that it stores the type qualifiers (`const`, `volatile`, `restrict`, plus some extended qualifiers required by language extensions) separately from the types themselves. `QualType` is conceptually a pair of "`Type*`" and the bits for these type qualifiers.

By storing the type qualifiers as bits in the conceptual pair, it is extremely efficient to get the set of qualifiers on a `QualType` (just return the field of the pair), add a type qualifier (which is a trivial constant-time operation that sets a bit), and remove one or more type qualifiers (just return a `QualType` with the bitfield set to empty).

Further, because the bits are stored outside of the type itself, we do not need to create duplicates of types with different sets of qualifiers (i.e. there is only a single heap allocated "`int`" type: "`const int`" and "`volatile`

`const int`" both point to the same heap allocated "`int`" type). This reduces the heap size used to represent bits and also means we do not have to consider qualifiers when uniquing types (Type does not even contain qualifiers).

In practice, the two most common type qualifiers (`const` and `restrict`) are stored in the low bits of the pointer to the `Type` object, together with a flag indicating whether extended qualifiers are present (which must be heap-allocated). This means that `QualType` is exactly the same size as a pointer.

Declaration names

The `DeclarationName` class represents the name of a declaration in Clang. Declarations in the C family of languages can take several different forms. Most declarations are named by simple identifiers, e.g., "`f`" and "`x`" in the function declaration `f(int x)`. In C++, declaration names can also name class constructors ("`Class`" in `struct Class { Class(); }`), class destructors ("`~Class`"), overloaded operator names ("`operator+`"), and conversion functions ("`operator void const *`"). In Objective-C, declaration names can refer to the names of Objective-C methods, which involve the method name and the parameters, collectively called a *selector*, e.g., "`setWidth:height:`". Since all of these kinds of entities — variables, functions, Objective-C methods, C++ constructors, destructors, and operators — are represented as subclasses of Clang's common `NamedDecl` class, `DeclarationName` is designed to efficiently represent any kind of name.

Given a `DeclarationName` `N`, `N.getNameKind()` will produce a value that describes what kind of name `N` stores. There are 10 options (all of the names are inside the `DeclarationName` class).

`Identifier`

> The name is a simple identifier. Use `N.getAsIdentifierInfo()` to retrieve the corresponding `IdentifierInfo*` pointing to the actual identifier.

`ObjCZeroArgSelector`, `ObjCOneArgSelector`, `ObjCMultiArgSelector`

> The name is an Objective-C selector, which can be retrieved as a `Selector` instance via `N.getObjCSelector()`. The three possible name kinds for Objective-C reflect an optimization within the `DeclarationName` class: both zero- and one-argument selectors are stored as a masked `IdentifierInfo` pointer, and therefore require very little space, since zero- and one-argument selectors are far more common than multi-argument selectors (which use a different structure).

`CXXConstructorName`

> The name is a C++ constructor name. Use `N.getCXXNameType()` to retrieve the type that this constructor is meant to construct. The type is always the canonical type, since all constructors for a given type have the same name.

`CXXDestructorName`

> The name is a C++ destructor name. Use `N.getCXXNameType()` to retrieve the type whose destructor is being named. This type is always a canonical type.

`CXXConversionFunctionName`

> The name is a C++ conversion function. Conversion functions are named according to the type they convert to, e.g., "`operator void const *`". Use `N.getCXXNameType()` to retrieve the type that this conversion function converts to. This type is always a canonical type.

`CXXOperatorName`

> The name is a C++ overloaded operator name. Overloaded operators are named according to their spelling, e.g., "`operator+`" or "`operator new []`". Use `N.getCXXOverloadedOperator()` to retrieve the overloaded operator (a value of type `OverloadedOperatorKind`).

`CXXLiteralOperatorName`

> The name is a C++11 user defined literal operator. User defined Literal operators are named according to the suffix they define, e.g., "`_foo`" for "`operator "" _foo`". Use `N.getCXXLiteralIdentifier()` to retrieve the corresponding `IdentifierInfo*` pointing to the identifier.

`CXXUsingDirective`

> The name is a C++ using directive. Using directives are not really NamedDecls, in that they all have the same name, but they are implemented as such in order to store them in DeclContext effectively.

`DeclarationNames` are cheap to create, copy, and compare. They require only a single pointer's worth of storage in the common cases (identifiers, zero- and one-argument Objective-C selectors) and use dense, uniqued storage for the other kinds of names. Two `DeclarationNames` can be compared for equality (==, !=) using a simple bitwise comparison, can be ordered with <, >, <=, and >= (which provide a lexicographical ordering for normal identifiers but an unspecified ordering for other kinds of names), and can be placed into LLVM `DenseMaps` and `DenseSets`.

`DeclarationName` instances can be created in different ways depending on what kind of name the instance will store. Normal identifiers (`IdentifierInfo` pointers) and Objective-C selectors (`Selector`) can be implicitly converted to `DeclarationNames`. Names for C++ constructors, destructors, conversion functions, and overloaded operators can be retrieved from the `DeclarationNameTable`, an instance of which is available as `ASTContext::DeclarationNames`. The member functions `getCXXConstructorName`, `getCXXDestructorName`, `getCXXConversionFunctionName`, and `getCXXOperatorName`, respectively, return `DeclarationName` instances for the four kinds of C++ special function names.

Declaration contexts

Every declaration in a program exists within some *declaration context*, such as a translation unit, namespace, class, or function. Declaration contexts in Clang are represented by the `DeclContext` class, from which the various declaration-context AST nodes (`TranslationUnitDecl`, `NamespaceDecl`, `RecordDecl`, `FunctionDecl`, etc.) will derive. The `DeclContext` class provides several facilities common to each declaration context:

Source-centric vs. Semantics-centric View of Declarations

> `DeclContext` provides two views of the declarations stored within a declaration context. The source-centric view accurately represents the program source code as written, including multiple declarations of entities where present (see the section Redeclarations and Overloads), while the semantics-centric view represents the program semantics. The two views are kept synchronized by semantic analysis while the ASTs are being constructed.

Storage of declarations within that context

> Every declaration context can contain some number of declarations. For example, a C++ class (represented by `RecordDecl`) contains various member functions, fields, nested types, and so on. All of these declarations will be stored within the `DeclContext`, and one can iterate over the declarations via [`DeclContext::decls_begin()`, `DeclContext::decls_end()`). This mechanism provides the source-centric view of declarations in the context.

Lookup of declarations within that context

> The `DeclContext` structure provides efficient name lookup for names within that declaration context. For example, if `N` is a namespace we can look for the name `N::f` using `DeclContext::lookup`. The lookup itself is based on a lazily-constructed array (for declaration contexts with a small number of declarations) or hash table (for declaration contexts with more declarations). The lookup operation provides the semantics-centric view of the declarations in the context.

Ownership of declarations

> The `DeclContext` owns all of the declarations that were declared within its declaration context, and is responsible for the management of their memory as well as their (de-)serialization.

All declarations are stored within a declaration context, and one can query information about the context in which each declaration lives. One can retrieve the `DeclContext` that contains a particular `Decl` using `Decl::getDeclContext`. However, see the section Lexical and Semantic Contexts for more information about how to interpret this context information.

Redeclarations and Overloads

Within a translation unit, it is common for an entity to be declared several times. For example, we might declare a function "f" and then later re-declare it as part of an inlined definition:

```
void f(int x, int y, int z = 1);

inline void f(int x, int y, int z) { /* ...  */ }
```

The representation of "f" differs in the source-centric and semantics-centric views of a declaration context. In the source-centric view, all redeclarations will be present, in the order they occurred in the source code, making this view

suitable for clients that wish to see the structure of the source code. In the semantics-centric view, only the most recent "f" will be found by the lookup, since it effectively replaces the first declaration of "f".

(Note that because f can be redeclared at block scope, or in a friend declaration, etc. it is possible that the declaration of f found by name lookup will not be the most recent one.)

In the semantics-centric view, overloading of functions is represented explicitly. For example, given two declarations of a function "g" that are overloaded, e.g.,

```
void g();
void g(int);
```

the DeclContext::lookup operation will return a DeclContext::lookup_result that contains a range of iterators over declarations of "g". Clients that perform semantic analysis on a program that is not concerned with the actual source code will primarily use this semantics-centric view.

Lexical and Semantic Contexts

Each declaration has two potentially different declaration contexts: a *lexical* context, which corresponds to the source-centric view of the declaration context, and a *semantic* context, which corresponds to the semantics-centric view. The lexical context is accessible via Decl::getLexicalDeclContext while the semantic context is accessible via Decl::getDeclContext, both of which return DeclContext pointers. For most declarations, the two contexts are identical. For example:

```
class X {
public:
  void f(int x);
};
```

Here, the semantic and lexical contexts of X::f are the DeclContext associated with the class X (itself stored as a RecordDecl AST node). However, we can now define X::f out-of-line:

```
void X::f(int x = 17) { /* ...  */ }
```

This definition of "f" has different lexical and semantic contexts. The lexical context corresponds to the declaration context in which the actual declaration occurred in the source code, e.g., the translation unit containing X. Thus, this declaration of X::f can be found by traversing the declarations provided by [decls_begin(), decls_end()) in the translation unit.

The semantic context of X::f corresponds to the class X, since this member function is (semantically) a member of X. Lookup of the name f into the DeclContext associated with X will then return the definition of X::f (including information about the default argument).

Transparent Declaration Contexts

In C and C++, there are several contexts in which names that are logically declared inside another declaration will actually "leak" out into the enclosing scope from the perspective of name lookup. The most obvious instance of this behavior is in enumeration types, e.g.,

```
enum Color {
  Red,
  Green,
  Blue
};
```

Here, Color is an enumeration, which is a declaration context that contains the enumerators Red, Green, and Blue. Thus, traversing the list of declarations contained in the enumeration Color will yield Red, Green, and Blue. However, outside of the scope of Color one can name the enumerator Red without qualifying the name, e.g.,

```
Color c = Red;
```

There are other entities in C++ that provide similar behavior. For example, linkage specifications that use curly braces:

```
extern "C" {
  void f(int);
```

```
  void g(int);
}
// f and g are visible here
```

For source-level accuracy, we treat the linkage specification and enumeration type as a declaration context in which its enclosed declarations ("`Red`", "`Green`", and "`Blue`"; "`f`" and "g") are declared. However, these declarations are visible outside of the scope of the declaration context.

These language features (and several others, described below) have roughly the same set of requirements: declarations are declared within a particular lexical context, but the declarations are also found via name lookup in scopes enclosing the declaration itself. This feature is implemented via *transparent* declaration contexts (see `DeclContext::isTransparentContext()`), whose declarations are visible in the nearest enclosing non-transparent declaration context. This means that the lexical context of the declaration (e.g., an enumerator) will be the transparent `DeclContext` itself, as will the semantic context, but the declaration will be visible in every outer context up to and including the first non-transparent declaration context (since transparent declaration contexts can be nested).

The transparent `DeclContexts` are:

- Enumerations (but not C++11 "scoped enumerations"):

```
enum Color {
  Red,
  Green,
  Blue
};
// Red, Green, and Blue are in scope
```

- C++ linkage specifications:

```
extern "C" {
  void f(int);
  void g(int);
}
// f and g are in scope
```

- Anonymous unions and structs:

```
struct LookupTable {
  bool IsVector;
  union {
    std::vector<Item> *Vector;
    std::set<Item> *Set;
  };
};

LookupTable LT;
LT.Vector = 0; // Okay: finds Vector inside the unnamed union
```

- C++11 inline namespaces:

```
namespace mylib {
  inline namespace debug {
    class X;
  }
}
mylib::X *xp; // okay: mylib::X refers to mylib::debug::X
```

Multiply-Defined Declaration Contexts

C++ namespaces have the interesting property that the namespace can be defined multiple times, and the declarations provided by each namespace definition are effectively merged (from the semantic point of view). For example, the following two code snippets are semantically indistinguishable:

```
// Snippet #1:
namespace N {
```

```
  void f();
}
namespace N {
  void f(int);
}

// Snippet #2:
namespace N {
  void f();
  void f(int);
}
```

In Clang's representation, the source-centric view of declaration contexts will actually have two separate `NamespaceDecl` nodes in Snippet #1, each of which is a declaration context that contains a single declaration of "`f`". However, the semantics-centric view provided by name lookup into the namespace `N` for "`f`" will return a `DeclContext::lookup_result` that contains a range of iterators over declarations of "`f`".

`DeclContext` manages multiply-defined declaration contexts internally. The function `DeclContext::getPrimaryContext` retrieves the "primary" context for a given `DeclContext` instance, which is the `DeclContext` responsible for maintaining the lookup table used for the semantics-centric view. Given a DeclContext, one can obtain the set of declaration contexts that are semantically connected to this declaration context, in source order, including this context (which will be the only result, for non-namespace contexts) via `DeclContext::collectAllContexts`. Note that these functions are used internally within the lookup and insertion methods of the `DeclContext`, so the vast majority of clients can ignore them.

Because the same entity can be defined multiple times in different modules, it is also possible for there to be multiple definitions of (for instance) a `CXXRecordDecl`, all of which describe a definition of the same class. In such a case, only one of those "definitions" is considered by Clang to be the definition of the class, and the others are treated as non-defining declarations that happen to also contain member declarations. Corresponding members in each definition of such multiply-defined classes are identified either by redeclaration chains (if the members are `Redeclarable`) or by simply a pointer to the canonical declaration (if the declarations are not `Redeclarable` – in that case, a `Mergeable` base class is used instead).

Error Handling

Clang produces an AST even when the code contains errors. Clang won't generate and optimize code for it, but it's used as parsing continues to detect further errors in the input. Clang-based tools also depend on such ASTs, and IDEs in particular benefit from a high-quality AST for broken code.

In presence of errors, clang uses a few error-recovery strategies to present the broken code in the AST:

- correcting errors: in cases where clang is confident about the fix, it provides a FixIt attaching to the error diagnostic and emits a corrected AST (reflecting the written code with FixIts applied). The advantage of that is to provide more accurate subsequent diagnostics. Typo correction is a typical example.
- representing invalid node: the invalid node is preserved in the AST in some form, e.g. when the "declaration" part of the declaration contains semantic errors, the Decl node is marked as invalid.
- dropping invalid node: this often happens for errors that we don't have graceful recovery. Prior to Recovery AST, a mismatched-argument function call expression was dropped though a CallExpr was created for semantic analysis.

With these strategies, clang surfaces better diagnostics, and provides AST consumers a rich AST reflecting the written source code as much as possible even for broken code.

Recovery AST

The idea of Recovery AST is to use recovery nodes which act as a placeholder to maintain the rough structure of the parsing tree, preserve locations and children but have no language semantics attached to them.

For example, consider the following mismatched function call:

```
int NoArg();
void test(int abc) {
```

```
  NoArg(abc); // oops, mismatched function arguments.
}
```

Without Recovery AST, the invalid function call expression (and its child expressions) would be dropped in the AST:

```
|-FunctionDecl <line:1:1, col:11> NoArg 'int ()'
`-FunctionDecl <line:2:1, line:4:1> test 'void (int)'
 |-ParmVarDecl <col:11, col:15> col:15 used abc 'int'
 `-CompoundStmt <col:20, line:4:1>
```

With Recovery AST, the AST looks like:

```
|-FunctionDecl <line:1:1, col:11> NoArg 'int ()'
`-FunctionDecl <line:2:1, line:4:1> test 'void (int)'
  |-ParmVarDecl <col:11, col:15> used abc 'int'
  `-CompoundStmt <col:20, line:4:1>
    `-RecoveryExpr <line:3:3, col:12> 'int' contains-errors
      |-UnresolvedLookupExpr <col:3> '<overloaded function type>' lvalue (ADL) = 'NoArg'
      `-DeclRefExpr <col:9> 'int' lvalue ParmVar 'abc' 'int'
```

An alternative is to use existing Exprs, e.g. CallExpr for the above example. This would capture more call details (e.g. locations of parentheses) and allow it to be treated uniformly with valid CallExprs. However, jamming the data we have into CallExpr forces us to weaken its invariants, e.g. arg count may be wrong. This would introduce a huge burden on consumers of the AST to handle such “impossible” cases. So when we’re representing (rather than correcting) errors, we use a distinct recovery node type with extremely weak invariants instead.

`RecoveryExpr` is the only recovery node so far. In practice, broken decls need more detailed semantics preserved (the current `Invalid` flag works fairly well), and completely broken statements with interesting internal structure are rare (so dropping the statements is OK).

Types and dependence

`RecoveryExpr` is an `Expr`, so it must have a type. In many cases the true type can’t really be known until the code is corrected (e.g. a call to a function that doesn’t exist). And it means that we can’t properly perform type checks on some containing constructs, such as `return 42 + unknownFunction()`.

To model this, we generalize the concept of dependence from C++ templates to mean dependence on a template parameter or how an error is repaired. The `RecoveryExpr unknownFunction()` has the totally unknown type `DependentTy`, and this suppresses type-based analysis in the same way it would inside a template.

In cases where we are confident about the concrete type (e.g. the return type for a broken non-overloaded function call), the `RecoveryExpr` will have this type. This allows more code to be typechecked, and produces a better AST and more diagnostics. For example:

```
unknownFunction().size() // .size() is a CXXDependentScopeMemberExpr
std::string(42).size() // .size() is a resolved MemberExpr
```

Whether or not the `RecoveryExpr` has a dependent type, it is always considered value-dependent, because its value isn’t well-defined until the error is resolved. Among other things, this means that clang doesn’t emit more errors where a RecoveryExpr is used as a constant (e.g. array size), but also won’t try to evaluate it.

ContainsErrors bit

Beyond the template dependence bits, we add a new “ContainsErrors” bit to express “Does this expression or anything within it contain errors” semantic, this bit is always set for RecoveryExpr, and propagated to other related nodes. This provides a fast way to query whether any (recursive) child of an expression had an error, which is often used to improve diagnostics.

```
// C++
void recoveryExpr(int abc) {
 unknownFunction(); // type-dependent, value-dependent, contains-errors

 std::string(42).size(); // value-dependent, contains-errors,
                         // not type-dependent, as we know the type is std::string
}
```

```
// C
void recoveryExpr(int abc) {
  unknownVar + abc; // type-dependent, value-dependent, contains-errors
}
```

The ASTImporter

The `ASTImporter` class imports nodes of an `ASTContext` into another `ASTContext`. Please refer to the document ASTImporter: Merging Clang ASTs for an introduction. And please read through the high-level description of the import algorithm, this is essential for understanding further implementation details of the importer.

Abstract Syntax Graph

Despite the name, the Clang AST is not a tree. It is a directed graph with cycles. One example of a cycle is the connection between a `ClassTemplateDecl` and its "templated" `CXXRecordDecl`. The *templated* `CXXRecordDecl` represents all the fields and methods inside the class template, while the `ClassTemplateDecl` holds the information which is related to being a template, i.e. template arguments, etc. We can get the *templated* class (the `CXXRecordDecl`) of a `ClassTemplateDecl` with `ClassTemplateDecl::getTemplatedDecl()`. And we can get back a pointer of the "described" class template from the *templated* class: `CXXRecordDecl::getDescribedTemplate()`. So, this is a cycle between two nodes: between the *templated* and the *described* node. There may be various other kinds of cycles in the AST especially in case of declarations.

Structural Equivalency

Importing one AST node copies that node into the destination `ASTContext`. To copy one node means that we create a new node in the "to" context then we set its properties to be equal to the properties of the source node. Before the copy, we make sure that the source node is not *structurally equivalent* to any existing node in the destination context. If it happens to be equivalent then we skip the copy.

The informal definition of structural equivalency is the following: Two nodes are **structurally equivalent** if they are

- builtin types and refer to the same type, e.g. `int` and `int` are structurally equivalent,
- function types and all their parameters have structurally equivalent types,
- record types and all their fields in order of their definition have the same identifier names and structurally equivalent types,
- variable or function declarations and they have the same identifier name and their types are structurally equivalent.

In C, two types are structurally equivalent if they are *compatible types*. For a formal definition of *compatible types*, please refer to 6.2.7/1 in the C11 standard. However, there is no definition for *compatible types* in the C++ standard. Still, we extend the definition of structural equivalency to templates and their instantiations similarly: besides checking the previously mentioned properties, we have to check for equivalent template parameters/arguments, etc.

The structural equivalent check can be and is used independently from the ASTImporter, e.g. the `clang::Sema` class uses it also.

The equivalence of nodes may depend on the equivalency of other pairs of nodes. Thus, the check is implemented as a parallel graph traversal. We traverse through the nodes of both graphs at the same time. The actual implementation is similar to breadth-first-search. Let's say we start the traverse with the <A,B> pair of nodes. Whenever the traversal reaches a pair <X,Y> then the following statements are true:

- A and X are nodes from the same ASTContext.
- B and Y are nodes from the same ASTContext.
- A and B may or may not be from the same ASTContext.
- if A == X and B == Y (pointer equivalency) then (there is a cycle during the traverse)
    - A and B are structurally equivalent if and only if
        - All dependent nodes on the path from <A,B> to <X,Y> are structurally equivalent.

When we compare two classes or enums and one of them is incomplete or has unloaded external lexical declarations then we cannot descend to compare their contained declarations. So in these cases they are

considered equal if they have the same names. This is the way how we compare forward declarations with definitions.

Redeclaration Chains

The early version of the `ASTImporter`'s merge mechanism squashed the declarations, i.e. it aimed to have only one declaration instead of maintaining a whole redeclaration chain. This early approach simply skipped importing a function prototype, but it imported a definition. To demonstrate the problem with this approach let's consider an empty "to" context and the following `virtual` function declarations of `f` in the "from" context:

```
struct B { virtual void f(); };
void B::f() {} // <-- let's import this definition
```

If we imported the definition with the "squashing" approach then we would end-up having one declaration which is indeed a definition, but `isVirtual()` returns `false` for it. The reason is that the definition is indeed not virtual, it is the property of the prototype!

Consequently, we must either set the virtual flag for the definition (but then we create a malformed AST which the parser would never create), or we import the whole redeclaration chain of the function. The most recent version of the `ASTImporter` uses the latter mechanism. We do import all function declarations - regardless if they are definitions or prototypes - in the order as they appear in the "from" context.

If we have an existing definition in the "to" context, then we cannot import another definition, we will use the existing definition. However, we can import prototype(s): we chain the newly imported prototype(s) to the existing definition. Whenever we import a new prototype from a third context, that will be added to the end of the redeclaration chain. This may result in long redeclaration chains in certain cases, e.g. if we import from several translation units which include the same header with the prototype.

To mitigate the problem of long redeclaration chains of free functions, we could compare prototypes to see if they have the same properties and if yes then we could merge these prototypes. The implementation of squashing of prototypes for free functions is future work.

Chaining functions this way ensures that we do copy all information from the source AST. Nonetheless, there is a problem with member functions: While we can have many prototypes for free functions, we must have only one prototype for a member function.

```
void f(); // OK
void f(); // OK

struct X {
  void f(); // OK
  void f(); // ERROR
};
void X::f() {} // OK
```

Thus, prototypes of member functions must be squashed, we cannot just simply attach a new prototype to the existing in-class prototype. Consider the following contexts:

```
// "to" context
struct X {
  void f(); // D0
};

// "from" context
struct X {
  void f(); // D1
};
void X::f() {} // D2
```

When we import the prototype and the definition of `f` from the "from" context, then the resulting redecl chain will look like this `D0 -> D2'`, where `D2'` is the copy of `D2` in the "to" context.

Generally speaking, when we import declarations (like enums and classes) we do attach the newly imported declaration to the existing redeclaration chain (if there is structural equivalency). We do not import, however, the whole redeclaration chain as we do in case of functions. Up till now, we haven't found any essential property of

forward declarations which is similar to the case of the virtual flag in a member function prototype. In the future, this may change, though.

Traversal during the Import

The node specific import mechanisms are implemented in `ASTNodeImporter::VisitNode()` functions, e.g. `VisitFunctionDecl()`. When we import a declaration then first we import everything which is needed to call the constructor of that declaration node. Everything which can be set later is set after the node is created. For example, in case of a `FunctionDecl` we first import the declaration context in which the function is declared, then we create the `FunctionDecl` and only then we import the body of the function. This means there are implicit dependencies between AST nodes. These dependencies determine the order in which we visit nodes in the "from" context. As with the regular graph traversal algorithms like DFS, we keep track which nodes we have already visited in `ASTImporter::ImportedDecls`. Whenever we create a node then we immediately add that to the `ImportedDecls`. We must not start the import of any other declarations before we keep track of the newly created one. This is essential, otherwise, we would not be able to handle circular dependencies. To enforce this, we wrap all constructor calls of all AST nodes in `GetImportedOrCreateDecl()`. This wrapper ensures that all newly created declarations are immediately marked as imported; also, if a declaration is already marked as imported then we just return its counterpart in the "to" context. Consequently, calling a declaration's `::Create()` function directly would lead to errors, please don't do that!

Even with the use of `GetImportedOrCreateDecl()` there is still a probability of having an infinite import recursion if things are imported from each other in wrong way. Imagine that during the import of `A`, the import of `B` is requested before we could create the node for `A` (the constructor needs a reference to `B`). And the same could be true for the import of `B` (`A` is requested to be imported before we could create the node for `B`). In case of the templated-described swing we take extra attention to break the cyclical dependency: we import and set the described template only after the `CXXRecordDecl` is created. As a best practice, before creating the node in the "to" context, avoid importing of other nodes which are not needed for the constructor of node `A`.

Error Handling

Every import function returns with either an `llvm::Error` or an `llvm::Expected<T>` object. This enforces to check the return value of the import functions. If there was an error during one import then we return with that error. (Exception: when we import the members of a class, we collect the individual errors with each member and we concatenate them in one Error object.) We cache these errors in cases of declarations. During the next import call if there is an existing error we just return with that. So, clients of the library receive an Error object, which they must check.

During import of a specific declaration, it may happen that some AST nodes had already been created before we recognize an error. In this case, we signal back the error to the caller, but the "to" context remains polluted with those nodes which had been created. Ideally, those nodes should not had been created, but that time we did not know about the error, the error happened later. Since the AST is immutable (most of the cases we can't remove existing nodes) we choose to mark these nodes as erroneous.

We cache the errors associated with declarations in the "from" context in `ASTImporter::ImportDeclErrors` and the ones which are associated with the "to" context in `ASTImporterSharedState::ImportErrors`. Note that, there may be several ASTImporter objects which import into the same "to" context but from different "from" contexts; in this case, they have to share the associated errors of the "to" context.

When an error happens, that propagates through the call stack, through all the dependant nodes. However, in case of dependency cycles, this is not enough, because we strive to mark the erroneous nodes so clients can act upon. In those cases, we have to keep track of the errors for those nodes which are intermediate nodes of a cycle.

An **import path** is the list of the AST nodes which we visit during an Import call. If node `A` depends on node `B` then the path contains an `A->B` edge. From the call stack of the import functions, we can read the very same path.

Now imagine the following AST, where the `->` represents dependency in terms of the import (all nodes are declarations).

```
A->B->C->D
   `->E
```

We would like to import A. The import behaves like a DFS, so we will visit the nodes in this order: ABCDE. During the visitation we will have the following import paths:

```
A
AB
ABC
ABCD
ABC
AB
ABE
AB
A
```

If during the visit of E there is an error then we set an error for E, then as the call stack shrinks for B, then for A:

```
A
AB
ABC
ABCD
ABC
AB
ABE // Error! Set an error to E
AB  // Set an error to B
A   // Set an error to A
```

However, during the import we could import C and D without any error and they are independent of A,B and E. We must not set up an error for C and D. So, at the end of the import we have an entry in `ImportDeclErrors` for A,B,E but not for C,D.

Now, what happens if there is a cycle in the import path? Let's consider this AST:

```
A->B->C->A
   `->E
```

During the visitation, we will have the below import paths and if during the visit of E there is an error then we will set up an error for E,B,A. But what's up with C?

```
A
AB
ABC
ABCA
ABC
AB
ABE // Error! Set an error to E
AB  // Set an error to B
A   // Set an error to A
```

This time we know that both B and C are dependent on A. This means we must set up an error for C too. As the call stack reverses back we get to A and we must set up an error to all nodes which depend on A (this includes C). But C is no longer on the import path, it just had been previously. Such a situation can happen only if during the visitation we had a cycle. If we didn't have any cycle, then the normal way of passing an Error object through the call stack could handle the situation. This is why we must track cycles during the import process for each visited declaration.

Lookup Problems

When we import a declaration from the source context then we check whether we already have a structurally equivalent node with the same name in the "to" context. If the "from" node is a definition and the found one is also a definition, then we do not create a new node, instead, we mark the found node as the imported node. If the found definition and the one we want to import have the same name but they are structurally in-equivalent, then we have an ODR violation in case of C++. If the "from" node is not a definition then we add that to the redeclaration chain of the found node. This behaviour is essential when we merge ASTs from different translation units which include the same header file(s). For example, we want to have only one definition for the class template `std::vector`, even if we included `<vector>` in several translation units.

To find a structurally equivalent node we can use the regular C/C++ lookup functions: `DeclContext::noload_lookup()` and `DeclContext::localUncachedLookup()`. These functions do respect the C/C++ name hiding rules, thus you cannot find certain declarations in a given declaration context. For

instance, unnamed declarations (anonymous structs), non-first `friend` declarations and template specializations are hidden. This is a problem, because if we use the regular C/C++ lookup then we create redundant AST nodes during the merge! Also, having two instances of the same node could result in false structural in-equivalencies of other nodes which depend on the duplicated node. Because of these reasons, we created a lookup class which has the sole purpose to register all declarations, so later they can be looked up by subsequent import requests. This is the `ASTImporterLookupTable` class. This lookup table should be shared amongst the different `ASTImporter` instances if they happen to import to the very same "to" context. This is why we can use the importer specific lookup only via the `ASTImporterSharedState` class.

ExternalASTSource

The `ExternalASTSource` is an abstract interface associated with the `ASTContext` class. It provides the ability to read the declarations stored within a declaration context either for iteration or for name lookup. A declaration context with an external AST source may load its declarations on-demand. This means that the list of declarations (represented as a linked list, the head is `DeclContext::FirstDecl`) could be empty. However, member functions like `DeclContext::lookup()` may initiate a load.

Usually, external sources are associated with precompiled headers. For example, when we load a class from a PCH then the members are loaded only if we do want to look up something in the class' context.

In case of LLDB, an implementation of the `ExternalASTSource` interface is attached to the AST context which is related to the parsed expression. This implementation of the `ExternalASTSource` interface is realized with the help of the `ASTImporter` class. This way, LLDB can reuse Clang's parsing machinery while synthesizing the underlying AST from the debug data (e.g. from DWARF). From the view of the `ASTImporter` this means both the "to" and the "from" context may have declaration contexts with external lexical storage. If a `DeclContext` in the "to" AST context has external lexical storage then we must take extra attention to work only with the already loaded declarations! Otherwise, we would end up with an uncontrolled import process. For instance, if we used the regular `DeclContext::lookup()` to find the existing declarations in the "to" context then the `lookup()` call itself would initiate a new import while we are in the middle of importing a declaration! (By the time we initiate the lookup we haven't registered yet that we already started to import the node of the "from" context.) This is why we use `DeclContext::noload_lookup()` instead.

Class Template Instantiations

Different translation units may have class template instantiations with the same template arguments, but with a different set of instantiated `MethodDecls` and `FieldDecls`. Consider the following files:

```
// x.h
template <typename T>
struct X {
    int a{0}; // FieldDecl with InitListExpr
    X(char) : a(3) {}     // (1)
    X(int) {}             // (2)
};

// foo.cpp
void foo() {
    // ClassTemplateSpec with ctor (1): FieldDecl without InitlistExpr
    X<char> xc('c');
}

// bar.cpp
void bar() {
    // ClassTemplateSpec with ctor (2): FieldDecl WITH InitlistExpr
    X<char> xc(1);
}
```

In `foo.cpp` we use the constructor with number `(1)`, which explicitly initializes the member `a` to 3, thus the `InitListExpr {0}` is not used here and the AST node is not instantiated. However, in the case of `bar.cpp` we use the constructor with number `(2)`, which does not explicitly initialize the `a` member, so the default `InitListExpr` is needed and thus instantiated. When we merge the AST of `foo.cpp` and `bar.cpp` we must create an AST node for the class template instantiation of `X<char>` which has all the required nodes. Therefore, when we find an existing `ClassTemplateSpecializationDecl` then we merge the fields of the

`ClassTemplateSpecializationDecl` in the “from” context in a way that the `InitListExpr` is copied if not existent yet. The same merge mechanism should be done in the cases of instantiated default arguments and exception specifications of functions.

Visibility of Declarations

During import of a global variable with external visibility, the lookup will find variables (with the same name) but with static visibility (linkage). Clearly, we cannot put them into the same redeclaration chain. The same is true the in case of functions. Also, we have to take care of other kinds of declarations like enums, classes, etc. if they are in anonymous namespaces. Therefore, we filter the lookup results and consider only those which have the same visibility as the declaration we currently import.

We consider two declarations in two anonymous namespaces to have the same visibility only if they are imported from the same AST context.

Strategies to Handle Conflicting Names

During the import we lookup existing declarations with the same name. We filter the lookup results based on their visibility. If any of the found declarations are not structurally equivalent then we bumped to a name conflict error (ODR violation in C++). In this case, we return with an `Error` and we set up the `Error` object for the declaration. However, some clients of the `ASTImporter` may require a different, perhaps less conservative and more liberal error handling strategy.

E.g. static analysis clients may benefit if the node is created even if there is a name conflict. During the CTU analysis of certain projects, we recognized that there are global declarations which collide with declarations from other translation units, but they are not referenced outside from their translation unit. These declarations should be in an unnamed namespace ideally. If we treat these collisions liberally then CTU analysis can find more results. Note, the feature be able to choose between name conflict handling strategies is still an ongoing work.

The `CFG` class

The `CFG` class is designed to represent a source-level control-flow graph for a single statement (`Stmt*`). Typically instances of `CFG` are constructed for function bodies (usually an instance of `CompoundStmt`), but can also be instantiated to represent the control-flow of any class that subclasses `Stmt`, which includes simple expressions. Control-flow graphs are especially useful for performing flow- or path-sensitive program analyses on a given function.

Basic Blocks

Concretely, an instance of `CFG` is a collection of basic blocks. Each basic block is an instance of `CFGBlock`, which simply contains an ordered sequence of `Stmt*` (each referring to statements in the AST). The ordering of statements within a block indicates unconditional flow of control from one statement to the next. Conditional control-flow is represented using edges between basic blocks. The statements within a given `CFGBlock` can be traversed using the `CFGBlock::*iterator` interface.

A `CFG` object owns the instances of `CFGBlock` within the control-flow graph it represents. Each `CFGBlock` within a CFG is also uniquely numbered (accessible via `CFGBlock::getBlockID()`). Currently the number is based on the ordering the blocks were created, but no assumptions should be made on how `CFGBlocks` are numbered other than their numbers are unique and that they are numbered from 0..N-1 (where N is the number of basic blocks in the CFG).

Entry and Exit Blocks

Each instance of `CFG` contains two special blocks: an *entry* block (accessible via `CFG::getEntry()`), which has no incoming edges, and an *exit* block (accessible via `CFG::getExit()`), which has no outgoing edges. Neither block contains any statements, and they serve the role of providing a clear entrance and exit for a body of code such as a function body. The presence of these empty blocks greatly simplifies the implementation of many analyses built on top of CFGs.

Conditional Control-Flow

Conditional control-flow (such as those induced by if-statements and loops) is represented as edges between `CFGBlocks`. Because different C language constructs can induce control-flow, each `CFGBlock` also records an extra `Stmt*` that represents the *terminator* of the block. A terminator is simply the statement that caused the control-flow, and is used to identify the nature of the conditional control-flow between blocks. For example, in the case of an if-statement, the terminator refers to the `IfStmt` object in the AST that represented the given branch.

To illustrate, consider the following code example:

```
int foo(int x) {
  x = x + 1;
  if (x > 2)
    x++;
  else {
    x += 2;
    x *= 2;
  }

  return x;
}
```

After invoking the parser+semantic analyzer on this code fragment, the AST of the body of `foo` is referenced by a single `Stmt*`. We can then construct an instance of `CFG` representing the control-flow graph of this function body by single call to a static class method:

```
Stmt *FooBody = ...
std::unique_ptr<CFG> FooCFG = CFG::buildCFG(FooBody);
```

Along with providing an interface to iterate over its `CFGBlocks`, the `CFG` class also provides methods that are useful for debugging and visualizing CFGs. For example, the method `CFG::dump()` dumps a pretty-printed version of the CFG to standard error. This is especially useful when one is using a debugger such as gdb. For example, here is the output of `FooCFG->dump()`:

```
[ B5 (ENTRY) ]
   Predecessors (0):
   Successors (1): B4

[ B4 ]
   1: x = x + 1
   2: (x > 2)
   T: if [B4.2]
   Predecessors (1): B5
   Successors (2): B3 B2

[ B3 ]
   1: x++
   Predecessors (1): B4
   Successors (1): B1

[ B2 ]
   1: x += 2
   2: x *= 2
   Predecessors (1): B4
   Successors (1): B1

[ B1 ]
   1: return x;
   Predecessors (2): B2 B3
   Successors (1): B0

[ B0 (EXIT) ]
```

```
Predecessors (1): B1
Successors (0):
```

For each block, the pretty-printed output displays for each block the number of *predecessor* blocks (blocks that have outgoing control-flow to the given block) and *successor* blocks (blocks that have control-flow that have incoming control-flow from the given block). We can also clearly see the special entry and exit blocks at the beginning and end of the pretty-printed output. For the entry block (block B5), the number of predecessor blocks is 0, while for the exit block (block B0) the number of successor blocks is 0.

The most interesting block here is B4, whose outgoing control-flow represents the branching caused by the sole if-statement in `foo`. Of particular interest is the second statement in the block, `(x > 2)`, and the terminator, printed as `if [B4.2]`. The second statement represents the evaluation of the condition of the if-statement, which occurs before the actual branching of control-flow. Within the `CFGBlock` for B4, the `Stmt*` for the second statement refers to the actual expression in the AST for `(x > 2)`. Thus pointers to subclasses of `Expr` can appear in the list of statements in a block, and not just subclasses of `Stmt` that refer to proper C statements.

The terminator of block B4 is a pointer to the `IfStmt` object in the AST. The pretty-printer outputs `if [B4.2]` because the condition expression of the if-statement has an actual place in the basic block, and thus the terminator is essentially *referring* to the expression that is the second statement of block B4 (i.e., B4.2). In this manner, conditions for control-flow (which also includes conditions for loops and switch statements) are hoisted into the actual basic block.

Constant Folding in the Clang AST

There are several places where constants and constant folding matter a lot to the Clang front-end. First, in general, we prefer the AST to retain the source code as close to how the user wrote it as possible. This means that if they wrote “5+4”, we want to keep the addition and two constants in the AST, we don’t want to fold to “9”. This means that constant folding in various ways turns into a tree walk that needs to handle the various cases.

However, there are places in both C and C++ that require constants to be folded. For example, the C standard defines what an “integer constant expression” (i-c-e) is with very precise and specific requirements. The language then requires i-c-e’s in a lot of places (for example, the size of a bitfield, the value for a case statement, etc). For these, we have to be able to constant fold the constants, to do semantic checks (e.g., verify bitfield size is non-negative and that case statements aren’t duplicated). We aim for Clang to be very pedantic about this, diagnosing cases when the code does not use an i-c-e where one is required, but accepting the code unless running with `-pedantic-errors`.

Things get a little bit more tricky when it comes to compatibility with real-world source code. Specifically, GCC has historically accepted a huge superset of expressions as i-c-e’s, and a lot of real world code depends on this unfortunate accident of history (including, e.g., the glibc system headers). GCC accepts anything its “fold” optimizer is capable of reducing to an integer constant, which means that the definition of what it accepts changes as its optimizer does. One example is that GCC accepts things like “`case X-X:`” even when `X` is a variable, because it can fold this to 0.

Another issue are how constants interact with the extensions we support, such as `__builtin_constant_p`, `__builtin_inf`, `__extension__` and many others. C99 obviously does not specify the semantics of any of these extensions, and the definition of i-c-e does not include them. However, these extensions are often used in real code, and we have to have a way to reason about them.

Finally, this is not just a problem for semantic analysis. The code generator and other clients have to be able to fold constants (e.g., to initialize global variables) and have to handle a superset of what C99 allows. Further, these clients can benefit from extended information. For example, we know that “`foo() || 1`” always evaluates to `true`, but we can’t replace the expression with `true` because it has side effects.

Implementation Approach

After trying several different approaches, we’ve finally converged on a design (Note, at the time of this writing, not all of this has been implemented, consider this a design goal!). Our basic approach is to define a single recursive evaluation method (`Expr::Evaluate`), which is implemented in `AST/ExprConstant.cpp`. Given an expression with “scalar” type (integer, fp, complex, or pointer) this method returns the following information:

- Whether the expression is an integer constant expression, a general constant that was folded but has no side effects, a general constant that was folded but that does have side effects, or an uncomputable/unfoldable value.

- If the expression was computable in any way, this method returns the `APValue` for the result of the expression.
- If the expression is not evaluatable at all, this method returns information on one of the problems with the expression. This includes a `SourceLocation` for where the problem is, and a diagnostic ID that explains the problem. The diagnostic should have `ERROR` type.
- If the expression is not an integer constant expression, this method returns information on one of the problems with the expression. This includes a `SourceLocation` for where the problem is, and a diagnostic ID that explains the problem. The diagnostic should have `EXTENSION` type.

This information gives various clients the flexibility that they want, and we will eventually have some helper methods for various extensions. For example, `Sema` should have a `Sema::VerifyIntegerConstantExpression` method, which calls `Evaluate` on the expression. If the expression is not foldable, the error is emitted, and it would return `true`. If the expression is not an i-c-e, the `EXTENSION` diagnostic is emitted. Finally it would return `false` to indicate that the AST is OK.

Other clients can use the information in other ways, for example, codegen can just use expressions that are foldable in any way.

Extensions

This section describes how some of the various extensions Clang supports interacts with constant evaluation:

- `__extension__`: The expression form of this extension causes any evaluatable subexpression to be accepted as an integer constant expression.
- `__builtin_constant_p`: This returns true (as an integer constant expression) if the operand evaluates to either a numeric value (that is, not a pointer cast to integral type) of integral, enumeration, floating or complex type, or if it evaluates to the address of the first character of a string literal (possibly cast to some other type). As a special case, if `__builtin_constant_p` is the (potentially parenthesized) condition of a conditional operator expression ("`?:`"), only the true side of the conditional operator is considered, and it is evaluated with full constant folding.
- `__builtin_choose_expr`: The condition is required to be an integer constant expression, but we accept any constant as an "extension of an extension". This only evaluates one operand depending on which way the condition evaluates.
- `__builtin_classify_type`: This always returns an integer constant expression.
- `__builtin_inf, nan, ...`: These are treated just like a floating-point literal.
- `__builtin_abs, copysign, ...`: These are constant folded as general constant expressions.
- `__builtin_strlen` and `strlen`: These are constant folded as integer constant expressions if the argument is a string literal.

The Sema Library

This library is called by the Parser library during parsing to do semantic analysis of the input. For valid programs, Sema builds an AST for parsed constructs.

The CodeGen Library

CodeGen takes an AST as input and produces LLVM IR code from it.

How to change Clang

How to add an attribute

Attributes are a form of metadata that can be attached to a program construct, allowing the programmer to pass semantic information along to the compiler for various uses. For example, attributes may be used to alter the code generation for a program construct, or to provide extra semantic information for static analysis. This document explains how to add a custom attribute to Clang. Documentation on existing attributes can be found here.

Attribute Basics

Attributes in Clang are handled in three stages: parsing into a parsed attribute representation, conversion from a parsed attribute into a semantic attribute, and then the semantic handling of the attribute.

Parsing of the attribute is determined by the various syntactic forms attributes can take, such as GNU, C++11, and Microsoft style attributes, as well as other information provided by the table definition of the attribute. Ultimately, the parsed representation of an attribute object is an `ParsedAttr` object. These parsed attributes chain together as a list of parsed attributes attached to a declarator or declaration specifier. The parsing of attributes is handled automatically by Clang, except for attributes spelled as keywords. When implementing a keyword attribute, the parsing of the keyword and creation of the `ParsedAttr` object must be done manually.

Eventually, `Sema::ProcessDeclAttributeList()` is called with a `Decl` and a `ParsedAttr`, at which point the parsed attribute can be transformed into a semantic attribute. The process by which a parsed attribute is converted into a semantic attribute depends on the attribute definition and semantic requirements of the attribute. The end result, however, is that the semantic attribute object is attached to the `Decl` object, and can be obtained by a call to `Decl::getAttr<T>()`. Similarly, for statement attributes, `Sema::ProcessStmtAttributes()` is called with a `Stmt` a list of `ParsedAttr` objects to be converted into a semantic attribute.

The structure of the semantic attribute is also governed by the attribute definition given in Attr.td. This definition is used to automatically generate functionality used for the implementation of the attribute, such as a class derived from `clang::Attr`, information for the parser to use, automated semantic checking for some attributes, etc.

`include/clang/Basic/Attr.td`

The first step to adding a new attribute to Clang is to add its definition to include/clang/Basic/Attr.td. This tablegen definition must derive from the `Attr` (tablegen, not semantic) type, or one of its derivatives. Most attributes will derive from the `InheritableAttr` type, which specifies that the attribute can be inherited by later redeclarations of the `Decl` it is associated with. `InheritableParamAttr` is similar to `InheritableAttr`, except that the attribute is written on a parameter instead of a declaration. If the attribute applies to statements, it should inherit from `StmtAttr`. If the attribute is intended to apply to a type instead of a declaration, such an attribute should derive from `TypeAttr`, and will generally not be given an AST representation. (Note that this document does not cover the creation of type attributes.) An attribute that inherits from `IgnoredAttr` is parsed, but will generate an ignored attribute diagnostic when used, which may be useful when an attribute is supported by another vendor but not supported by clang.

The definition will specify several key pieces of information, such as the semantic name of the attribute, the spellings the attribute supports, the arguments the attribute expects, and more. Most members of the `Attr` tablegen type do not require definitions in the derived definition as the default suffice. However, every attribute must specify at least a spelling list, a subject list, and a documentation list.

Spellings

All attributes are required to specify a spelling list that denotes the ways in which the attribute can be spelled. For instance, a single semantic attribute may have a keyword spelling, as well as a C++11 spelling and a GNU spelling. An empty spelling list is also permissible and may be useful for attributes which are created implicitly. The following spellings are accepted:

Spelling	Description
`GNU`	Spelled with a GNU-style `__attribute__((attr))` syntax and placement.
`CXX11`	Spelled with a C++-style `[[attr]]` syntax with an optional vendor-specific namespace.
`C2x`	Spelled with a C-style `[[attr]]` syntax with an optional vendor-specific namespace.
`Declspec`	Spelled with a Microsoft-style `__declspec(attr)` syntax.
`Keyword`	The attribute is spelled as a keyword, and required custom parsing.
`GCC`	Specifies two or three spellings: the first is a GNU-style spelling, the second is a C++-style spelling with the `gnu` namespace, and the third is an optional C-style spelling with the `gnu` namespace. Attributes should only specify this spelling for attributes supported by GCC.

Spelling	Description
`Clang`	Specifies two or three spellings: the first is a GNU-style spelling, the second is a C++-style spelling with the `clang` namespace, and the third is an optional C-style spelling with the `clang` namespace. By default, a C-style spelling is provided.
`Pragma`	The attribute is spelled as a `#pragma`, and requires custom processing within the preprocessor. If the attribute is meant to be used by Clang, it should set the namespace to `"clang"`. Note that this spelling is not used for declaration attributes.

Subjects

Attributes appertain to one or more subjects. If the attribute attempts to attach to a subject that is not in the subject list, a diagnostic is issued automatically. Whether the diagnostic is a warning or an error depends on how the attribute's `SubjectList` is defined, but the default behavior is to warn. The diagnostics displayed to the user are automatically determined based on the subjects in the list, but a custom diagnostic parameter can also be specified in the `SubjectList`. The diagnostics generated for subject list violations are calculated automatically or specified by the subject list itself. If a previously unused Decl node is added to the `SubjectList`, the logic used to automatically determine the diagnostic parameter in utils/TableGen/ClangAttrEmitter.cpp may need to be updated.

By default, all subjects in the SubjectList must either be a Decl node defined in `DeclNodes.td`, or a statement node defined in `StmtNodes.td`. However, more complex subjects can be created by creating a `SubsetSubject` object. Each such object has a base subject which it appertains to (which must be a Decl or Stmt node, and not a SubsetSubject node), and some custom code which is called when determining whether an attribute appertains to the subject. For instance, a `NonBitField` SubsetSubject appertains to a `FieldDecl`, and tests whether the given FieldDecl is a bit field. When a SubsetSubject is specified in a SubjectList, a custom diagnostic parameter must also be provided.

Diagnostic checking for attribute subject lists for declaration and statement attributes is automated except when `HasCustomParsing` is set to `1`.

Documentation

All attributes must have some form of documentation associated with them. Documentation is table generated on the public web server by a server-side process that runs daily. Generally, the documentation for an attribute is a stand-alone definition in include/clang/Basic/AttrDocs.td that is named after the attribute being documented.

If the attribute is not for public consumption, or is an implicitly-created attribute that has no visible spelling, the documentation list can specify the `Undocumented` object. Otherwise, the attribute should have its documentation added to AttrDocs.td.

Documentation derives from the `Documentation` tablegen type. All derived types must specify a documentation category and the actual documentation itself. Additionally, it can specify a custom heading for the attribute, though a default heading will be chosen when possible.

There are four predefined documentation categories: `DocCatFunction` for attributes that appertain to function-like subjects, `DocCatVariable` for attributes that appertain to variable-like subjects, `DocCatType` for type attributes, and `DocCatStmt` for statement attributes. A custom documentation category should be used for groups of attributes with similar functionality. Custom categories are good for providing overview information for the attributes grouped under it. For instance, the consumed annotation attributes define a custom category, `DocCatConsumed`, that explains what consumed annotations are at a high level.

Documentation content (whether it is for an attribute or a category) is written using reStructuredText (RST) syntax.

After writing the documentation for the attribute, it should be locally tested to ensure that there are no issues generating the documentation on the server. Local testing requires a fresh build of clang-tblgen. To generate the attribute documentation, execute the following command:

```
clang-tblgen -gen-attr-docs -I /path/to/clang/include /path/to/clang/include/clang/Basic/Attr.td -o /path/to/clang/docs/AttributeReference.rst
```

When testing locally, *do not* commit changes to `AttributeReference.rst`. This file is generated by the server automatically, and any changes made to this file will be overwritten.

Arguments

Attributes may optionally specify a list of arguments that can be passed to the attribute. Attribute arguments specify both the parsed form and the semantic form of the attribute. For example, if `Args` is `[StringArgument<"Arg1">, IntArgument<"Arg2">]` then `__attribute__((myattribute("Hello", 3)))` will be a valid use; it requires two arguments while parsing, and the Attr subclass' constructor for the semantic attribute will require a string and integer argument.

All arguments have a name and a flag that specifies whether the argument is optional. The associated C++ type of the argument is determined by the argument definition type. If the existing argument types are insufficient, new types can be created, but it requires modifying utils/TableGen/ClangAttrEmitter.cpp to properly support the type.

Other Properties

The `Attr` definition has other members which control the behavior of the attribute. Many of them are special-purpose and beyond the scope of this document, however a few deserve mention.

If the parsed form of the attribute is more complex, or differs from the semantic form, the `HasCustomParsing` bit can be set to `1` for the class, and the parsing code in Parser::ParseGNUAttributeArgs() can be updated for the special case. Note that this only applies to arguments with a GNU spelling – attributes with a __declspec spelling currently ignore this flag and are handled by `Parser::ParseMicrosoftDeclSpec`.

Note that setting this member to 1 will opt out of common attribute semantic handling, requiring extra implementation efforts to ensure the attribute appertains to the appropriate subject, etc.

If the attribute should not be propagated from a template declaration to an instantiation of the template, set the `Clone` member to 0. By default, all attributes will be cloned to template instantiations.

Attributes that do not require an AST node should set the `ASTNode` field to `0` to avoid polluting the AST. Note that anything inheriting from `TypeAttr` or `IgnoredAttr` automatically do not generate an AST node. All other attributes generate an AST node by default. The AST node is the semantic representation of the attribute.

The `LangOpts` field specifies a list of language options required by the attribute. For instance, all of the CUDA-specific attributes specify `[CUDA]` for the `LangOpts` field, and when the CUDA language option is not enabled, an "attribute ignored" warning diagnostic is emitted. Since language options are not table generated nodes, new language options must be created manually and should specify the spelling used by `LangOptions` class.

Custom accessors can be generated for an attribute based on the spelling list for that attribute. For instance, if an attribute has two different spellings: 'Foo' and 'Bar', accessors can be created: `[Accessor<"isFoo", [GNU<"Foo">]>, Accessor<"isBar", [GNU<"Bar">]>]` These accessors will be generated on the semantic form of the attribute, accepting no arguments and returning a `bool`.

Attributes that do not require custom semantic handling should set the `SemaHandler` field to `0`. Note that anything inheriting from `IgnoredAttr` automatically do not get a semantic handler. All other attributes are assumed to use a semantic handler by default. Attributes without a semantic handler are not given a parsed attribute `Kind` enumerator.

"Simple" attributes, that require no custom semantic processing aside from what is automatically provided, should set the `SimpleHandler` field to `1`.

Target-specific attributes may share a spelling with other attributes in different targets. For instance, the ARM and MSP430 targets both have an attribute spelled `GNU<"interrupt">`, but with different parsing and semantic requirements. To support this feature, an attribute inheriting from `TargetSpecificAttribute` may specify a `ParseKind` field. This field should be the same value between all arguments sharing a spelling, and corresponds to the parsed attribute's `Kind` enumerator. This allows attributes to share a parsed attribute kind, but have distinct semantic attribute classes. For instance, `ParsedAttr` is the shared parsed attribute kind, but ARMInterruptAttr and MSP430InterruptAttr are the semantic attributes generated.

By default, attribute arguments are parsed in an evaluated context. If the arguments for an attribute should be parsed in an unevaluated context (akin to the way the argument to a `sizeof` expression is parsed), set `ParseArgumentsAsUnevaluated` to `1`.

If additional functionality is desired for the semantic form of the attribute, the `AdditionalMembers` field specifies code to be copied verbatim into the semantic attribute class object, with `public` access.

If two or more attributes cannot be used in combination on the same declaration or statement, a `MutualExclusions` definition can be supplied to automatically generate diagnostic code. This will disallow the attribute combinations regardless of spellings used. Additionally, it will diagnose combinations within the same attribute list, different attribute list, and redeclarations, as appropriate.

Boilerplate

All semantic processing of declaration attributes happens in lib/Sema/SemaDeclAttr.cpp, and generally starts in the `ProcessDeclAttribute()` function. If the attribute has the `SimpleHandler` field set to `1` then the function to process the attribute will be automatically generated, and nothing needs to be done here. Otherwise, write a new `handleYourAttr()` function, and add that to the switch statement. Please do not implement handling logic directly in the `case` for the attribute.

Unless otherwise specified by the attribute definition, common semantic checking of the parsed attribute is handled automatically. This includes diagnosing parsed attributes that do not appertain to the given `Decl` or `Stmt`, ensuring the correct minimum number of arguments are passed, etc.

If the attribute adds additional warnings, define a `DiagGroup` in include/clang/Basic/DiagnosticGroups.td named after the attribute's `Spelling` with "_"s replaced by "-"s. If there is only a single diagnostic, it is permissible to use `InGroup<DiagGroup<"your-attribute">>` directly in DiagnosticSemaKinds.td

All semantic diagnostics generated for your attribute, including automatically- generated ones (such as subjects and argument counts), should have a corresponding test case.

Semantic handling

Most attributes are implemented to have some effect on the compiler. For instance, to modify the way code is generated, or to add extra semantic checks for an analysis pass, etc. Having added the attribute definition and conversion to the semantic representation for the attribute, what remains is to implement the custom logic requiring use of the attribute.

The `clang::Decl` object can be queried for the presence or absence of an attribute using `hasAttr<T>()`. To obtain a pointer to the semantic representation of the attribute, `getAttr<T>` may be used.

The `clang::AttributedStmt` object can be queried for the presence or absence of an attribute by calling `getAttrs()` and looping over the list of attributes.

How to add an expression or statement

Expressions and statements are one of the most fundamental constructs within a compiler, because they interact with many different parts of the AST, semantic analysis, and IR generation. Therefore, adding a new expression or statement kind into Clang requires some care. The following list details the various places in Clang where an expression or statement needs to be introduced, along with patterns to follow to ensure that the new expression or statement works well across all of the C languages. We focus on expressions, but statements are similar.

1. Introduce parsing actions into the parser. Recursive-descent parsing is mostly self-explanatory, but there are a few things that are worth keeping in mind:
    - Keep as much source location information as possible! You'll want it later to produce great diagnostics and support Clang's various features that map between source code and the AST.
    - Write tests for all of the "bad" parsing cases, to make sure your recovery is good. If you have matched delimiters (e.g., parentheses, square brackets, etc.), use `Parser::BalancedDelimiterTracker` to give nice diagnostics when things go wrong.
2. Introduce semantic analysis actions into `Sema`. Semantic analysis should always involve two functions: an `ActOnXXX` function that will be called directly from the parser, and a `BuildXXX` function that performs the actual semantic analysis and will (eventually!) build the AST node. It's fairly common for the `ActOnCXX` function to do very little (often just some minor translation from the parser's representation to `Sema`'s representation of the same thing), but the separation is still important: C++ template instantiation, for example, should always call the `BuildXXX` variant. Several notes on semantic analysis before we get into construction of the AST:
    - Your expression probably involves some types and some subexpressions. Make sure to fully check that those types, and the types of those subexpressions, meet your expectations. Add implicit conversions where necessary to make sure that all of the types line up exactly the way you want them. Write extensive tests to check that you're getting good diagnostics for mistakes and that you can use various forms of subexpressions with your expression.
    - When type-checking a type or subexpression, make sure to first check whether the type is "dependent" (`Type::isDependentType()`) or whether a subexpression is type-dependent

(`Expr::isTypeDependent()`). If any of these return `true`, then you're inside a template and you can't do much type-checking now. That's normal, and your AST node (when you get there) will have to deal with this case. At this point, you can write tests that use your expression within templates, but don't try to instantiate the templates.

- For each subexpression, be sure to call `Sema::CheckPlaceholderExpr()` to deal with "weird" expressions that don't behave well as subexpressions. Then, determine whether you need to perform lvalue-to-rvalue conversions (`Sema::DefaultLvalueConversions`) or the usual unary conversions (`Sema::UsualUnaryConversions`), for places where the subexpression is producing a value you intend to use.
- Your `BuildXXX` function will probably just return `ExprError()` at this point, since you don't have an AST. That's perfectly fine, and shouldn't impact your testing.

3. Introduce an AST node for your new expression. This starts with declaring the node in `include/Basic/StmtNodes.td` and creating a new class for your expression in the appropriate `include/AST/Expr*.h` header. It's best to look at the class for a similar expression to get ideas, and there are some specific things to watch for:

   - If you need to allocate memory, use the `ASTContext` allocator to allocate memory. Never use raw `malloc` or `new`, and never hold any resources in an AST node, because the destructor of an AST node is never called.
   - Make sure that `getSourceRange()` covers the exact source range of your expression. This is needed for diagnostics and for IDE support.
   - Make sure that `children()` visits all of the subexpressions. This is important for a number of features (e.g., IDE support, C++ variadic templates). If you have sub-types, you'll also need to visit those sub-types in `RecursiveASTVisitor`.
   - Add printing support (`StmtPrinter.cpp`) for your expression.
   - Add profiling support (`StmtProfile.cpp`) for your AST node, noting the distinguishing (non-source location) characteristics of an instance of your expression. Omitting this step will lead to hard-to-diagnose failures regarding matching of template declarations.
   - Add serialization support (`ASTReaderStmt.cpp`, `ASTWriterStmt.cpp`) for your AST node.

4. Teach semantic analysis to build your AST node. At this point, you can wire up your `Sema::BuildXXX` function to actually create your AST. A few things to check at this point:

   - If your expression can construct a new C++ class or return a new Objective-C object, be sure to update and then call `Sema::MaybeBindToTemporary` for your just-created AST node to be sure that the object gets properly destructed. An easy way to test this is to return a C++ class with a private destructor: semantic analysis should flag an error here with the attempt to call the destructor.
   - Inspect the generated AST by printing it using `clang -cc1 -ast-print`, to make sure you're capturing all of the important information about how the AST was written.
   - Inspect the generated AST under `clang -cc1 -ast-dump` to verify that all of the types in the generated AST line up the way you want them. Remember that clients of the AST should never have to "think" to understand what's going on. For example, all implicit conversions should show up explicitly in the AST.
   - Write tests that use your expression as a subexpression of other, well-known expressions. Can you call a function using your expression as an argument? Can you use the ternary operator?

5. Teach code generation to create IR to your AST node. This step is the first (and only) that requires knowledge of LLVM IR. There are several things to keep in mind:

   - Code generation is separated into scalar/aggregate/complex and lvalue/rvalue paths, depending on what kind of result your expression produces. On occasion, this requires some careful factoring of code to avoid duplication.
   - `CodeGenFunction` contains functions `ConvertType` and `ConvertTypeForMem` that convert Clang's types (`clang::Type*` or `clang::QualType`) to LLVM types. Use the former for values, and the latter for memory locations: test with the C++ "`bool`" type to check this. If you find that you are having to use LLVM bitcasts to make the subexpressions of your expression have the type that your expression expects, STOP! Go fix semantic analysis and the AST so that you don't need these bitcasts.

- The `CodeGenFunction` class has a number of helper functions to make certain operations easy, such as generating code to produce an lvalue or an rvalue, or to initialize a memory location with a given value. Prefer to use these functions rather than directly writing loads and stores, because these functions take care of some of the tricky details for you (e.g., for exceptions).
- If your expression requires some special behavior in the event of an exception, look at the `push*Cleanup` functions in `CodeGenFunction` to introduce a cleanup. You shouldn't have to deal with exception-handling directly.
- Testing is extremely important in IR generation. Use `clang -cc1 -emit-llvm` and FileCheck to verify that you're generating the right IR.

6. Teach template instantiation how to cope with your AST node, which requires some fairly simple code:

- Make sure that your expression's constructor properly computes the flags for type dependence (i.e., the type your expression produces can change from one instantiation to the next), value dependence (i.e., the constant value your expression produces can change from one instantiation to the next), instantiation dependence (i.e., a template parameter occurs anywhere in your expression), and whether your expression contains a parameter pack (for variadic templates). Often, computing these flags just means combining the results from the various types and subexpressions.
- Add `TransformXXX` and `RebuildXXX` functions to the `TreeTransform` class template in `Sema`. `TransformXXX` should (recursively) transform all of the subexpressions and types within your expression, using `getDerived().TransformYYY`. If all of the subexpressions and types transform without error, it will then call the `RebuildXXX` function, which will in turn call `getSema().BuildXXX` to perform semantic analysis and build your expression.
- To test template instantiation, take those tests you wrote to make sure that you were type checking with type-dependent expressions and dependent types (from step #2) and instantiate those templates with various types, some of which type-check and some that don't, and test the error messages in each case.

7. There are some "extras" that make other features work better. It's worth handling these extras to give your expression complete integration into Clang:

- Add code completion support for your expression in `SemaCodeComplete.cpp`.
- If your expression has types in it, or has any "interesting" features other than subexpressions, extend libclang's `CursorVisitor` to provide proper visitation for your expression, enabling various IDE features such as syntax highlighting, cross-referencing, and so on. The `c-index-test` helper program can be used to test these features.

Driver Design & Internals

Introduction 966
Features and Goals 966
GCC Compatibility 966
Flexible 966
Low Overhead 966
Simple 966
Internal Design and Implementation 966
Internals Introduction 967
Design Overview 967
Driver Stages 968
Additional Notes 971
The Compilation Object 971
Unified Parsing & Pipelining 971
ToolChain Argument Translation 971
Unused Argument Warnings 971
Relation to GCC Driver Concepts 971

Introduction

This document describes the Clang driver. The purpose of this document is to describe both the motivation and design goals for the driver, as well as details of the internal implementation.

Features and Goals

The Clang driver is intended to be a production quality compiler driver providing access to the Clang compiler and tools, with a command line interface which is compatible with the gcc driver.

Although the driver is part of and driven by the Clang project, it is logically a separate tool which shares many of the same goals as Clang:

GCC Compatibility 966
Flexible 966
Low Overhead 966
Simple 966

GCC Compatibility

The number one goal of the driver is to ease the adoption of Clang by allowing users to drop Clang into a build system which was designed to call GCC. Although this makes the driver much more complicated than might otherwise be necessary, we decided that being very compatible with the gcc command line interface was worth it in order to allow users to quickly test clang on their projects.

Flexible

The driver was designed to be flexible and easily accommodate new uses as we grow the clang and LLVM infrastructure. As one example, the driver can easily support the introduction of tools which have an integrated assembler; something we hope to add to LLVM in the future.

Similarly, most of the driver functionality is kept in a library which can be used to build other tools which want to implement or accept a gcc like interface.

Low Overhead

The driver should have as little overhead as possible. In practice, we found that the gcc driver by itself incurred a small but meaningful overhead when compiling many small files. The driver doesn't do much work compared to a compilation, but we have tried to keep it as efficient as possible by following a few simple principles:

- Avoid memory allocation and string copying when possible.
- Don't parse arguments more than once.
- Provide a few simple interfaces for efficiently searching arguments.

Simple

Finally, the driver was designed to be "as simple as possible", given the other goals. Notably, trying to be completely compatible with the gcc driver adds a significant amount of complexity. However, the design of the driver attempts to mitigate this complexity by dividing the process into a number of independent stages instead of a single monolithic task.

Internal Design and Implementation

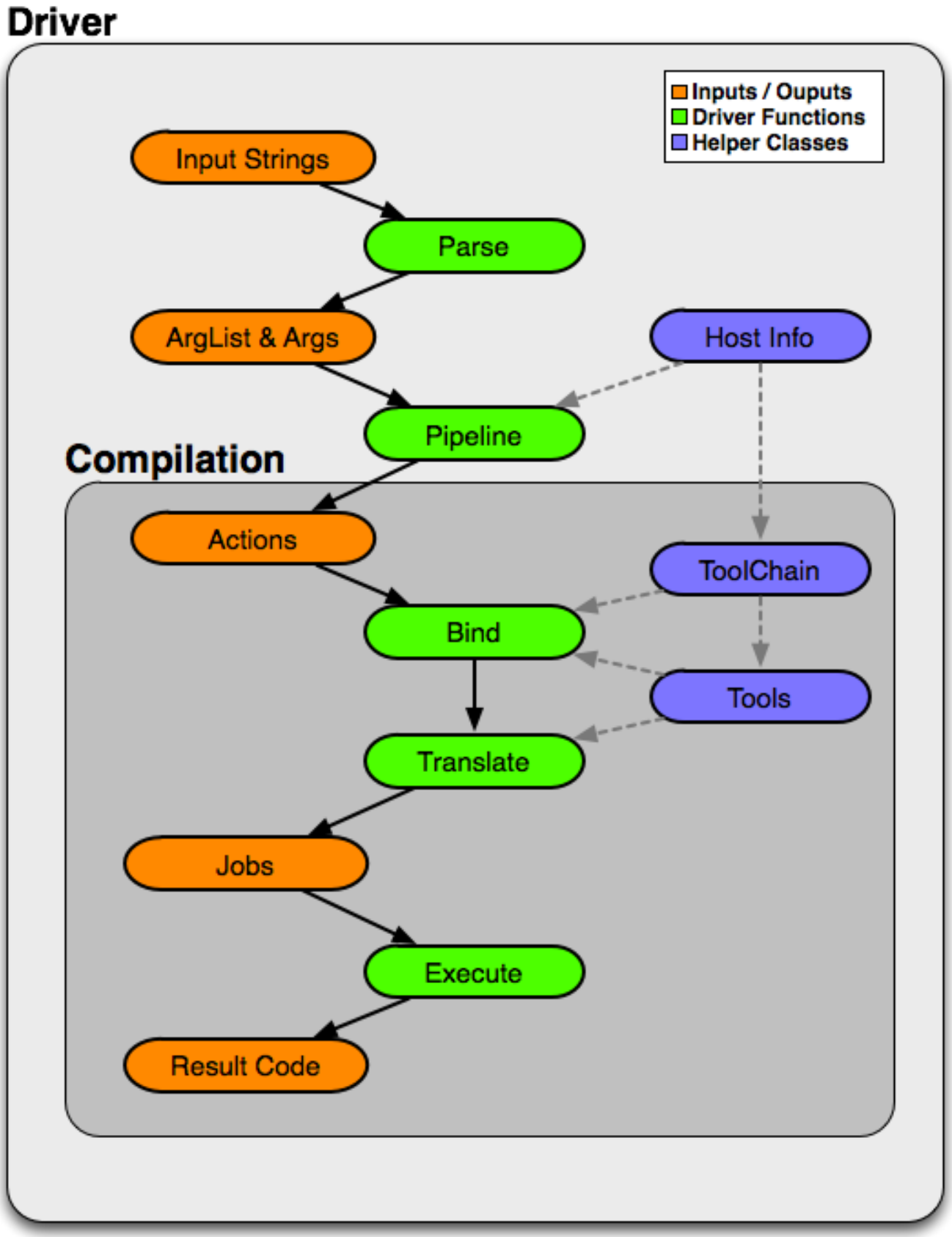
Internals Introduction 967
Design Overview 967
Driver Stages 968
Additional Notes 971
Relation to GCC Driver Concepts 971

Internals Introduction

In order to satisfy the stated goals, the driver was designed to completely subsume the functionality of the gcc executable; that is, the driver should not need to delegate to gcc to perform subtasks. On Darwin, this implies that the Clang driver also subsumes the gcc driver-driver, which is used to implement support for building universal images (binaries and object files). This also implies that the driver should be able to call the language specific compilers (e.g. cc1) directly, which means that it must have enough information to forward command line arguments to child processes correctly.

Design Overview

The diagram below shows the significant components of the driver architecture and how they relate to one another. The orange components represent concrete data structures built by the driver, the green components indicate conceptually distinct stages which manipulate these data structures, and the blue components are important helper classes.



Driver Stages

The driver functionality is conceptually divided into five stages:

1. **Parse: Option Parsing**

The command line argument strings are decomposed into arguments (`Arg` instances). The driver expects to understand all available options, although there is some facility for just passing certain classes of options through (like `-Wl,`).

Each argument corresponds to exactly one abstract `Option` definition, which describes how the option is parsed along with some additional metadata. The Arg instances themselves are lightweight and merely contain enough information for clients to determine which option they correspond to and their values (if they have additional parameters).

For example, a command line like “-Ifoo -I foo” would parse to two Arg instances (a JoinedArg and a SeparateArg instance), but each would refer to the same Option.

Options are lazily created in order to avoid populating all Option classes when the driver is loaded. Most of the driver code only needs to deal with options by their unique ID (e.g., `options::OPT_I`),

Arg instances themselves do not generally store the values of parameters. In many cases, this would simply result in creating unnecessary string copies. Instead, Arg instances are always embedded inside an ArgList structure, which contains the original vector of argument strings. Each Arg itself only needs to contain an index into this vector instead of storing its values directly.

The clang driver can dump the results of this stage using the `-###` flag (which must precede any actual command line arguments). For example:

```
$ clang -### -Xarch_i386 -fomit-frame-pointer -Wa,-fast -Ifoo -I foo t.c
Option 0 - Name: "-Xarch_", Values: {"i386", "-fomit-frame-pointer"}
Option 1 - Name: "-Wa,", Values: {"-fast"}
Option 2 - Name: "-I", Values: {"foo"}
Option 3 - Name: "-I", Values: {"foo"}
Option 4 - Name: "<input>", Values: {"t.c"}
```

After this stage is complete the command line should be broken down into well defined option objects with their appropriate parameters. Subsequent stages should rarely, if ever, need to do any string processing.

2. **Pipeline: Compilation Action Construction**

Once the arguments are parsed, the tree of subprocess jobs needed for the desired compilation sequence are constructed. This involves determining the input files and their types, what work is to be done on them (preprocess, compile, assemble, link, etc.), and constructing a list of Action instances for each task. The result is a list of one or more top-level actions, each of which generally corresponds to a single output (for example, an object or linked executable).

The majority of Actions correspond to actual tasks, however there are two special Actions. The first is InputAction, which simply serves to adapt an input argument for use as an input to other Actions. The second is BindArchAction, which conceptually alters the architecture to be used for all of its input Actions.

The clang driver can dump the results of this stage using the `-ccc-print-phases` flag. For example:

```
$ clang -ccc-print-phases -x c t.c -x assembler t.s
0: input, "t.c", c
1: preprocessor, {0}, cpp-output
2: compiler, {1}, assembler
3: assembler, {2}, object
4: input, "t.s", assembler
5: assembler, {4}, object
6: linker, {3, 5}, image
```

Here the driver is constructing seven distinct actions, four to compile the “t.c” input into an object file, two to assemble the “t.s” input, and one to link them together.

A rather different compilation pipeline is shown here; in this example there are two top level actions to compile the input files into two separate object files, where each object file is built using `lipo` to merge results built for two separate architectures.

```
$ clang -ccc-print-phases -c -arch i386 -arch x86_64 t0.c t1.c
0: input, "t0.c", c
1: preprocessor, {0}, cpp-output
```

```
2: compiler, {1}, assembler
3: assembler, {2}, object
4: bind-arch, "i386", {3}, object
5: bind-arch, "x86_64", {3}, object
6: lipo, {4, 5}, object
7: input, "t1.c", c
8: preprocessor, {7}, cpp-output
9: compiler, {8}, assembler
10: assembler, {9}, object
11: bind-arch, "i386", {10}, object
12: bind-arch, "x86_64", {10}, object
13: lipo, {11, 12}, object
```

After this stage is complete the compilation process is divided into a simple set of actions which need to be performed to produce intermediate or final outputs (in some cases, like `-fsyntax-only`, there is no “real” final output). Phases are well known compilation steps, such as “preprocess”, “compile”, “assemble”, “link”, etc.

3. **Bind: Tool & Filename Selection**

   This stage (in conjunction with the Translate stage) turns the tree of Actions into a list of actual subprocess to run. Conceptually, the driver performs a top down matching to assign Action(s) to Tools. The ToolChain is responsible for selecting the tool to perform a particular action; once selected the driver interacts with the tool to see if it can match additional actions (for example, by having an integrated preprocessor).

   Once Tools have been selected for all actions, the driver determines how the tools should be connected (for example, using an inprocess module, pipes, temporary files, or user provided filenames). If an output file is required, the driver also computes the appropriate file name (the suffix and file location depend on the input types and options such as `-save-temps`).

   The driver interacts with a ToolChain to perform the Tool bindings. Each ToolChain contains information about all the tools needed for compilation for a particular architecture, platform, and operating system. A single driver invocation may query multiple ToolChains during one compilation in order to interact with tools for separate architectures.

   The results of this stage are not computed directly, but the driver can print the results via the `-ccc-print-bindings` option. For example:

```
$ clang -ccc-print-bindings -arch i386 -arch ppc t0.c
# "i386-apple-darwin9" - "clang", inputs: ["t0.c"], output: "/tmp/cc-Sn4RKF.s"
# "i386-apple-darwin9" - "darwin::Assemble", inputs: ["/tmp/cc-Sn4RKF.s"], output: "/tmp/cc-gvSnbS.o"
# "i386-apple-darwin9" - "darwin::Link", inputs: ["/tmp/cc-gvSnbS.o"], output: "/tmp/cc-jgHQxi.out"
# "ppc-apple-darwin9" - "gcc::Compile", inputs: ["t0.c"], output: "/tmp/cc-Q0bTox.s"
# "ppc-apple-darwin9" - "gcc::Assemble", inputs: ["/tmp/cc-Q0bTox.s"], output: "/tmp/cc-WCdicw.o"
# "ppc-apple-darwin9" - "gcc::Link", inputs: ["/tmp/cc-WCdicw.o"], output: "/tmp/cc-HHBEBh.out"
# "i386-apple-darwin9" - "darwin::Lipo", inputs: ["/tmp/cc-jgHQxi.out", "/tmp/cc-HHBEBh.out"], output: "a.out"
```

   This shows the tool chain, tool, inputs and outputs which have been bound for this compilation sequence. Here clang is being used to compile t0.c on the i386 architecture and darwin specific versions of the tools are being used to assemble and link the result, but generic gcc versions of the tools are being used on PowerPC.

4. **Translate: Tool Specific Argument Translation**

   Once a Tool has been selected to perform a particular Action, the Tool must construct concrete Commands which will be executed during compilation. The main work is in translating from the gcc style command line options to whatever options the subprocess expects.

   Some tools, such as the assembler, only interact with a handful of arguments and just determine the path of the executable to call and pass on their input and output arguments. Others, like the compiler or the linker, may translate a large number of arguments in addition.

   The ArgList class provides a number of simple helper methods to assist with translating arguments; for example, to pass on only the last of arguments corresponding to some option, or all arguments for an option.

   The result of this stage is a list of Commands (executable paths and argument strings) to execute.

5. **Execute**
   Finally, the compilation pipeline is executed. This is mostly straightforward, although there is some interaction with options like `-pipe`, `-pass-exit-codes` and `-time`.

Additional Notes

The Compilation Object

The driver constructs a Compilation object for each set of command line arguments. The Driver itself is intended to be invariant during construction of a Compilation; an IDE should be able to construct a single long lived driver instance to use for an entire build, for example.

The Compilation object holds information that is particular to each compilation sequence. For example, the list of used temporary files (which must be removed once compilation is finished) and result files (which should be removed if compilation fails).

Unified Parsing & Pipelining

Parsing and pipelining both occur without reference to a Compilation instance. This is by design; the driver expects that both of these phases are platform neutral, with a few very well defined exceptions such as whether the platform uses a driver driver.

ToolChain Argument Translation

In order to match gcc very closely, the clang driver currently allows tool chains to perform their own translation of the argument list (into a new ArgList data structure). Although this allows the clang driver to match gcc easily, it also makes the driver operation much harder to understand (since the Tools stop seeing some arguments the user provided, and see new ones instead).

For example, on Darwin `-gfull` gets translated into two separate arguments, `-g` and `-fno-eliminate-unused-debug-symbols`. Trying to write Tool logic to do something with `-gfull` will not work, because Tool argument translation is done after the arguments have been translated.

A long term goal is to remove this tool chain specific translation, and instead force each tool to change its own logic to do the right thing on the untranslated original arguments.

Unused Argument Warnings

The driver operates by parsing all arguments but giving Tools the opportunity to choose which arguments to pass on. One downside of this infrastructure is that if the user misspells some option, or is confused about which options to use, some command line arguments the user really cared about may go unused. This problem is particularly important when using clang as a compiler, since the clang compiler does not support anywhere near all the options that gcc does, and we want to make sure users know which ones are being used.

To support this, the driver maintains a bit associated with each argument of whether it has been used (at all) during the compilation. This bit usually doesn't need to be set by hand, as the key ArgList accessors will set it automatically.

When a compilation is successful (there are no errors), the driver checks the bit and emits an "unused argument" warning for any arguments which were never accessed. This is conservative (the argument may not have been used to do what the user wanted) but still catches the most obvious cases.

Relation to GCC Driver Concepts

For those familiar with the gcc driver, this section provides a brief overview of how things from the gcc driver map to the clang driver.

- **Driver Driver**

  The driver driver is fully integrated into the clang driver. The driver simply constructs additional Actions to bind the architecture during the *Pipeline* phase. The tool chain specific argument translation is responsible for handling `-Xarch_`.

  The one caveat is that this approach requires `-Xarch_` not be used to alter the compilation itself (for example, one cannot provide `-S` as an `-Xarch_` argument). The driver attempts to reject such invocations, and overall there isn't a good reason to abuse `-Xarch_` to that end in practice.

  The upside is that the clang driver is more efficient and does little extra work to support universal builds. It also provides better error reporting and UI consistency.

- **Specs**

  The clang driver has no direct correspondent for “specs”. The majority of the functionality that is embedded in specs is in the Tool specific argument translation routines. The parts of specs which control the compilation pipeline are generally part of the *Pipeline* stage.

- **Toolchains**

  The gcc driver has no direct understanding of tool chains. Each gcc binary roughly corresponds to the information which is embedded inside a single ToolChain.

  The clang driver is intended to be portable and support complex compilation environments. All platform and tool chain specific code should be protected behind either abstract or well defined interfaces (such as whether the platform supports use as a driver driver).

Offloading Design & Internals

Introduction 972

OpenMP Offloading 972

Offloading Overview 972

Compilation Process 973

Generating Offloading Entries 973

Accessing Entries on the Device 974

Debugging Information 974

Offload Device Compilation 974

Creating Fat Objects 975

Linking Target Device Code 975

Device Binary Wrapping 975

Structure Types 975

Global Variables 976

Binary Descriptor for Device Images 976

Global Constructor and Destructor 977

Offloading Example 977

Introduction

This document describes the Clang driver and code generation steps for creating offloading applications. Clang supports offloading to various architectures using programming models like CUDA, HIP, and OpenMP. The purpose of this document is to illustrate the steps necessary to create an offloading application using Clang.

OpenMP Offloading

Clang supports OpenMP target offloading to several different architectures such as NVPTX, AMDGPU, X86_64, Arm, and PowerPC. Offloading code is generated by Clang and then executed using the `libomptarget` runtime and the associated plugin for the target architecture, e.g. `libomptarget.rtl.cuda`. This section describes the steps necessary to create a functioning device image that can be loaded by the OpenMP runtime. More information on the OpenMP runtimes can be found at the OpenMP documentation page.

Offloading Overview

The goal of offloading compilation is to create an executable device image that can be run on the target device. OpenMP offloading creates executable images by compiling the input file for both the host and the target device. The output from the device phase then needs to be embedded into the host to create a fat object. A special tool then needs to extract the device code from the fat objects, run the device linking step, and embed the final image in a symbol the host runtime library can use to register the library and access the symbols on the device.

Compilation Process

The compiler performs the following high-level actions to generate OpenMP offloading code:

- Compile the input file for the host to produce a bitcode file. Lower `#pragma omp target` declarations to offloading entries and create metadata to indicate which entries are on the device.
- Compile the input file for the target device using the offloading entry metadata created by the host.
- Link the OpenMP device runtime library and run the backend to create a device object file.
- Run the backend on the host bitcode file and create a fat object file using the device object file.
- Pass the fat object file to the linker wrapper tool and extract the device objects. Run the device linking action on the extracted objects.
- Wrap the device images and offload entries in a symbol that can be accessed by the host.
- Add the wrapped binary to the linker input and run the host linking action. Link with `libomptarget` to register and execute the images.

Generating Offloading Entries

The first step in compilation is to generate offloading entries for the host. This information is used to identify function kernels or global values that will be provided by the device. Blocks contained in a `#pragma omp target` or symbols inside a `#pragma omp declare target` directive will have offloading entries generated. The following table shows the offload entry structure.

__tgt_offload_entry Structure

Type	Identifier	Description
void*	addr	Address of global symbol within device image (function or global)
char*	name	Name of the symbol
size_t	size	Size of the entry info (0 if it is a function)
int32_t	flags	Flags associated with the entry (see Target Region Entry Flags)
int32_t	reserved	Reserved, to be used by the runtime library.

The address of the global symbol will be set to the device pointer value by the runtime once the device image is loaded. The flags are set to indicate the handling required for the offloading entry. If the offloading entry is an entry to a target region it can have one of the following entry flags.

Target Region Entry Flags

Name	Value	Description
OMPTargetRegionEntryTargetRegion	0x00	Mark the entry as generic target region
OMPTargetRegionEntryCtor	0x02	Mark the entry as a global constructor
OMPTargetRegionEntryDtor	0x04	Mark the entry as a global destructor

If the offloading entry is a global variable, indicated by a non-zero size, it will instead have one of the following global flags.

Target Region Global

Name	Value	Description
OMPTargetGlobalVarEntryTo	0x00	Mark the entry as a ‘to’ attribute (w.r.t. the to clause)
OMPTargetGlobalVarEntryLink	0x01	Mark the entry as a ‘link’ attribute (w.r.t. the link clause)

The target offload entries are used by the runtime to access the device kernels and globals that will be provided by the final device image. Each offloading entry is set to use the `omp_offloading_entries` section. When the final application is created the linker will provide the `__start_omp_offloading_entries` and `__stop_omp_offloading_entries` symbols which are used to create the final image.

This information is used by the device compilation stage to determine which symbols need to be exported from the device. We use the `omp_offload.info` metadata node to pass this information device compilation stage.

Accessing Entries on the Device

Accessing the entries in the device is done using the address field in the offload entry. The runtime will set the address to the pointer associated with the device image during runtime initialization. This is used to call the corresponding kernel function when entering a `#pragma omp target` region. For variables, the runtime maintains a table mapping host pointers to device pointers. Global variables inside a `#pragma omp target declare` directive are first initialized to the host’s address. Once the device address is initialized we insert it into the table to map the host address to the device address.

Debugging Information

We generate structures to hold debugging information that is passed to `libomptarget`. This allows the front-end to generate information the runtime library uses for more informative error messages. This is done using the standard identifier structure used in `libomp` and `libomptarget`. This is used to pass information and source locations to the runtime.

ident_t Structure

Type	Identifier	Description
int32_t	reserved	Reserved, to be used by the runtime library.
int32_t	flags	Flags used to indicate some features, mostly unused.
int32_t	reserved	Reserved, to be used by the runtime library.
int32_t	reserved	Reserved, to be used by the runtime library.
char*	psource	Program source information, stored as “;filename;function;line;column;;\0”

If debugging information is enabled, we will also create strings to indicate the names and declarations of variables mapped in target regions. These have the same format as the source location in the identifier structure, but the function name is replaced with the variable name.

Offload Device Compilation

The input file is compiled for each active device toolchain. The device compilation stage is performed differently from the host stage. Namely, we do not generate any offloading entries. This is set by passing the `-fopenmp-is-device` flag to the front-end. We use the host bitcode to determine which symbols to export from the device. The bitcode file is passed in from the previous stage using the `-fopenmp-host-ir-file-path` flag. Compilation is otherwise performed as it would be for any other target triple.

When compiling for the OpenMP device, we set the visibility of all device symbols to be `protected` by default. This improves performance and prevents a class of errors where a symbol in the target device could preempt a host library.

The OpenMP runtime library is linked in during compilation to provide the implementations for standard OpenMP functionality. For GPU targets this is done by linking in a special bitcode library during compilation, (e.g. `libomptarget-nvptx64-sm_70.bc`) using the `-mlink-builtin-bitcode` flag. Other device libraries, such as

CUDA's libdevice, are also linked this way. If the target is a standard architecture with an existing `libomp` implementation, that will be linked instead. Finally, device tools are used to create a relocatable device object file that can be embedded in the host.

Creating Fat Objects

A fat binary is a binary file that contains information intended for another device. We create a fat object by embedding the output of the device compilation stage into the host as a named section. The output from the device compilation is passed to the host backend using the `-fembed-offload-object` flag. This embeds the device image into the `.llvm.offloading` section using a special binary format that behaves like a string map. This binary format is used to bundle metadata about the image so the linker can associate the proper device linking action with the image. Each device image will start with the magic bytes `0x10FF10AD`.

```
@llvm.embedded.object = private constant [1 x i8] c"\00", section ".llvm.offloading"
```

The device code will then be placed in the corresponding section one the backend is run on the host, creating a fat object. Using fat objects allows us to treat offloading objects as standard host objects. The final object file should contain the following offloading sections. We will use this information when Linking Target Device Code.

Offloading Sections

Section	Description
omp_offloading_entries	Offloading entry information (see __tgt_offload_entry structure)
.llvm.offloading	Embedded device object file for the target device and architecture

Linking Target Device Code

Objects containing Offloading Sections require special handling to create an executable device image. This is done using a Clang tool, see Clang Linker Wrapper for more information. This tool works as a wrapper over the host linking job. It scans the input object files for the offloading section `.llvm.offloading`. The device files stored in this section are then extracted and passed tot he appropriate linking job. The linked device image is then wrapped to create the symbols used to load the device image and link it with the host.

The linker wrapper tool supports linking bitcode files through link time optimization (LTO). This is used whenever the object files embedded in the host contain LLVM bitcode. Bitcode will be embedded for architectures that do not support a relocatable object format, such as AMDGPU or SPIR-V, or if the user requested it using the `-foffload-lto` flag.

Device Binary Wrapping

Various structures and functions are used to create the information necessary to offload code on the device. We use the linked device executable with the corresponding offloading entries to create the symbols necessary to load and execute the device image.

Structure Types

Several different structures are used to store offloading information. The device image structure stores a single linked device image and its associated offloading entries. The offloading entries are stored using the `__start_omp_offloading_entries` and `__stop_omp_offloading_entries` symbols generated by the linker using the __tgt_offload_entry structure.

__tgt_device_image Structure

Type	Identifier	Description
void*	ImageStart	Pointer to the target code start
void*	ImageEnd	Pointer to the target code end
__tgt_offload_entry*	EntriesBegin	Begin of table with all target entries

Type	Identifier	Description
__tgt_offload_entry*	EntriesEnd	End of table (non inclusive)

The target target binary descriptor is used to store all binary images and offloading entries in an array.

__tgt_bin_desc Structure

Type	Identifier	Description
int32_t	NumDeviceImages	Number of device types supported
__tgt_device_image*	DeviceImages	Array of device images (1 per dev. type)
__tgt_offload_entry*	HostEntriesBegin	Begin of table with all host entries
__tgt_offload_entry*	HostEntriesEnd	End of table (non inclusive)

Global Variables

Global Variables lists various global variables, along with their type and their explicit ELF sections, which are used to store device images and related symbols.

Global Variables

Variable	Type	ELF Section	Description
__start_omp_offloading_entries	__tgt_offload_entry	.omp_offloading_entries	Begin symbol for the offload entries table.
__stop_omp_offloading_entries	__tgt_offload_entry	.omp_offloading_entries	End symbol for the offload entries table.
__dummy.omp_offloading.entry	__tgt_offload_entry	.omp_offloading_entries	Dummy zero-sized object in the offload entries section to force linker to define begin/end symbols defined above.
.omp_offloading.device_image	__tgt_device_image	.omp_offloading_entries	ELF device code object of the first image.
.omp_offloading.device_image.N	__tgt_device_image	.omp_offloading_entries	ELF device code object of the (N+1)th image.
.omp_offloading.device_images	__tgt_device_image	.omp_offloading_entries	Array of images.
.omp_offloading.descriptor	__tgt_bin_desc	.omp_offloading_entries	Binary descriptor object (see Binary Descriptor for Device Images)

Binary Descriptor for Device Images

This object is passed to the offloading runtime at program startup and it describes all device images available in the executable or shared library. It is defined as follows:

```
__attribute__((visibility("hidden")))
extern __tgt_offload_entry *__start_omp_offloading_entries;
__attribute__((visibility("hidden")))
extern __tgt_offload_entry *__stop_omp_offloading_entries;
static const char Image0[] = { <Bufs.front() contents> };
...
static const char ImageN[] = { <Bufs.back() contents> };
static const __tgt_device_image Images[] = {
  {
    Image0,                                 /*ImageStart*/
```

```
    Image0 + sizeof(Image0),           /*ImageEnd*/
    __start_omp_offloading_entries,    /*EntriesBegin*/
    __stop_omp_offloading_entries      /*EntriesEnd*/
  },
  ...
  {
    ImageN,                            /*ImageStart*/
    ImageN + sizeof(ImageN),           /*ImageEnd*/
    __start_omp_offloading_entries,    /*EntriesBegin*/
    __stop_omp_offloading_entries      /*EntriesEnd*/
  }
};
static const __tgt_bin_desc BinDesc = {
  sizeof(Images) / sizeof(Images[0]),  /*NumDeviceImages*/
  Images,                              /*DeviceImages*/
  __start_omp_offloading_entries,      /*HostEntriesBegin*/
  __stop_omp_offloading_entries        /*HostEntriesEnd*/
};
```

Global Constructor and Destructor

The global constructor (`.omp_offloading.descriptor_reg()`) registers the device images with the runtime by calling the `__tgt_register_lib()` runtime function. The constructor is explicitly defined in `.text.startup` section and is run once when the program starts. Similarly, the global destructor (`.omp_offloading.descriptor_unreg()`) calls `__tgt_unregister_lib()` for the destructor and is also defined in `.text.startup` section and run when the program exits.

Offloading Example

This section contains a simple example of generating offloading code using OpenMP offloading. We will use a simple ZAXPY BLAS routine.

```
#include <complex>

using complex = std::complex<double>;

void zaxpy(complex *X, complex *Y, complex D, std::size_t N) {
#pragma omp target teams distribute parallel for
  for (std::size_t i = 0; i < N; ++i)
    Y[i] = D * X[i] + Y[i];
}

int main() {
  const std::size_t N = 1024;
  complex X[N], Y[N], D;
#pragma omp target data map(to:X[0 : N]) map(tofrom:Y[0 : N])
  zaxpy(X, Y, D, N);
}
```

This code is compiled using the following Clang flags.

```
$ clang++ -fopenmp -fopenmp-targets=nvptx64 -O3 zaxpy.cpp -c
```

The output section in the object file can be seen using the `readelf` utility. The `.llvm.offloading` section has the `SHF_EXCLUDE` flag so it will be removed from the final executable or shared library by the linker.

```
$ llvm-readelf -WS zaxpy.o
Section Headers:
[Nr] Name                   Type     Address          Off    Size   ES Flg Lk Inf Al
[11] omp_offloading_entries PROGBITS 0000000000000000 0001f0 000040 00   A  0   0  1
[12] .llvm.offloading       PROGBITS 0000000000000000 000260 030950 00   E  0   0  8
```

Compiling this file again will invoke the `clang-linker-wrapper` utility to extract and link the device code stored at the section named `.llvm.offloading` and then use entries stored in the section named `omp_offloading_entries` to create the symbols necessary for `libomptarget` to register the device image and call the entry function.

```
$ clang++ -fopenmp -fopenmp-targets=nvptx64 zaxpy.o -o zaxpy
$ ./zaxpy
```

We can see the steps created by clang to generate the offloading code using the `-ccc-print-phases` option in Clang. This matches the description in Offloading Overview.

```
$ clang++ -fopenmp -fopenmp-targets=nvptx64 -ccc-print-phases zaxpy.cpp
# "x86_64-unknown-linux-gnu" - "clang", inputs: ["zaxpy.cpp"], output: "/tmp/zaxpy-host.bc"
# "nvptx64-nvidia-cuda" - "clang", inputs: ["zaxpy.cpp", "/tmp/zaxpy-e6a41b.bc"], output: "/tmp/zaxpy-07f434.s"
# "nvptx64-nvidia-cuda" - "NVPTX::Assembler", inputs: ["/tmp/zaxpy-07f434.s"], output: "/tmp/zaxpy-0af7b7.o"
# "x86_64-unknown-linux-gnu" - "clang", inputs: ["/tmp/zaxpy-e6a41b.bc", "/tmp/zaxpy-0af7b7.o"], output: "/tmp/zaxpy-416cad.o"
# "x86_64-unknown-linux-gnu" - "Offload::Linker", inputs: ["/tmp/zaxpy-416cad.o"], output: "a.out"
```

Precompiled Header and Modules Internals

Using Precompiled Headers with `clang` 978

Design Philosophy 978

AST File Contents 979

- Metadata Block 980
- Source Manager Block 981
- Preprocessor Block 981
- Types Block 981
- Declarations Block 981
- Statements and Expressions 982
- Identifier Table Block 982
- Method Pool Block 983

AST Reader Integration Points 983

Chained precompiled headers 984

Modules 984

This document describes the design and implementation of Clang's precompiled headers (PCH) and modules. If you are interested in the end-user view, please see the User's Manual.

Using Precompiled Headers with `clang`

The Clang compiler frontend, `clang -cc1`, supports two command line options for generating and using PCH files.

To generate PCH files using `clang -cc1`, use the option *-emit-pch*:

```
$ clang -cc1 test.h -emit-pch -o test.h.pch
```

This option is transparently used by `clang` when generating PCH files. The resulting PCH file contains the serialized form of the compiler's internal representation after it has completed parsing and semantic analysis. The PCH file can then be used as a prefix header with the *-include-pch* option:

```
$ clang -cc1 -include-pch test.h.pch test.c -o test.s
```

Design Philosophy

Precompiled headers are meant to improve overall compile times for projects, so the design of precompiled headers is entirely driven by performance concerns. The use case for precompiled headers is relatively simple: when there is a common set of headers that is included in nearly every source file in the project, we *precompile* that bundle of headers into a single precompiled header (PCH file). Then, when compiling the source files in the project, we load the PCH file first (as a prefix header), which acts as a stand-in for that bundle of headers.

A precompiled header implementation improves performance when:

- Loading the PCH file is significantly faster than re-parsing the bundle of headers stored within the PCH file. Thus, a precompiled header design attempts to minimize the cost of reading the PCH file. Ideally, this cost should not vary with the size of the precompiled header file.
- The cost of generating the PCH file initially is not so large that it counters the per-source-file performance improvement due to eliminating the need to parse the bundled headers in the first place. This is particularly important on multi-core systems, because PCH file generation serializes the build when all compilations require the PCH file to be up-to-date.

Modules, as implemented in Clang, use the same mechanisms as precompiled headers to save a serialized AST file (one per module) and use those AST modules. From an implementation standpoint, modules are a generalization of precompiled headers, lifting a number of restrictions placed on precompiled headers. In particular, there can only be one precompiled header and it must be included at the beginning of the translation unit. The extensions to the AST file format required for modules are discussed in the section on modules.

Clang's AST files are designed with a compact on-disk representation, which minimizes both creation time and the time required to initially load the AST file. The AST file itself contains a serialized representation of Clang's abstract syntax trees and supporting data structures, stored using the same compressed bitstream as LLVM's bitcode file format.

Clang's AST files are loaded "lazily" from disk. When an AST file is initially loaded, Clang reads only a small amount of data from the AST file to establish where certain important data structures are stored. The amount of data read in this initial load is independent of the size of the AST file, such that a larger AST file does not lead to longer AST load times. The actual header data in the AST file — macros, functions, variables, types, etc. — is loaded only when it is referenced from the user's code, at which point only that entity (and those entities it depends on) are deserialized from the AST file. With this approach, the cost of using an AST file for a translation unit is proportional to the amount of code actually used from the AST file, rather than being proportional to the size of the AST file itself.

When given the *-print-stats* option, Clang produces statistics describing how much of the AST file was actually loaded from disk. For a simple "Hello, World!" program that includes the Apple `Cocoa.h` header (which is built as a precompiled header), this option illustrates how little of the actual precompiled header is required:

```
*** AST File Statistics:
  895/39981 source location entries read (2.238563%)
  19/15315 types read (0.124061%)
  20/82685 declarations read (0.024188%)
  154/58070 identifiers read (0.265197%)
  0/7260 selectors read (0.000000%)
  0/30842 statements read (0.000000%)
  4/8400 macros read (0.047619%)
  1/4995 lexical declcontexts read (0.020020%)
  0/4413 visible declcontexts read (0.000000%)
  0/7230 method pool entries read (0.000000%)
  0 method pool misses
```

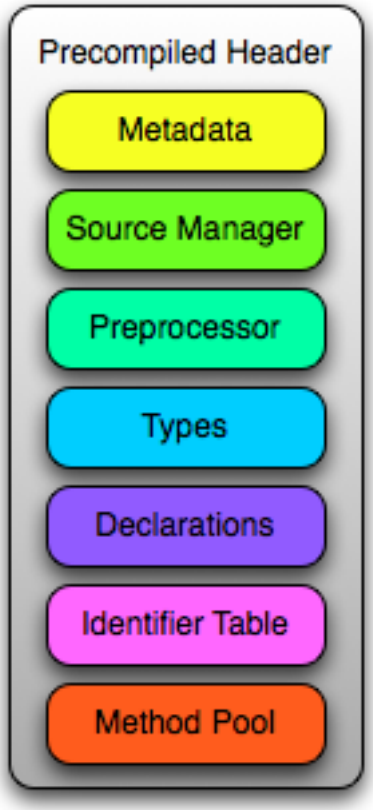
For this small program, only a tiny fraction of the source locations, types, declarations, identifiers, and macros were actually deserialized from the precompiled header. These statistics can be useful to determine whether the AST file implementation can be improved by making more of the implementation lazy.

Precompiled headers can be chained. When you create a PCH while including an existing PCH, Clang can create the new PCH by referencing the original file and only writing the new data to the new file. For example, you could create a PCH out of all the headers that are very commonly used throughout your project, and then create a PCH for every single source file in the project that includes the code that is specific to that file, so that recompiling the file itself is very fast, without duplicating the data from the common headers for every file. The mechanisms behind chained precompiled headers are discussed in a later section.

AST File Contents

An AST file produced by clang is an object file container with a `clangast` (COFF) or `__clangast` (ELF and Mach-O) section containing the serialized AST. Other target-specific sections in the object file container are used to hold debug information for the data types defined in the AST. Tools built on top of libclang that do not need debug information may also produce raw AST files that only contain the serialized AST.

The `clangast` section is organized into several different blocks, each of which contains the serialized representation of a part of Clang's internal representation. Each of the blocks corresponds to either a block or a record within LLVM's bitstream format. The contents of each of these logical blocks are described below.



The `llvm-objdump` utility provides a `-raw-clang-ast` option to extract the binary contents of the AST section from an object file container.

The llvm-bcanalyzer utility can be used to examine the actual structure of the bitstream for the AST section. This information can be used both to help understand the structure of the AST section and to isolate areas where the AST representation can still be optimized, e.g., through the introduction of abbreviations.

Metadata Block

The metadata block contains several records that provide information about how the AST file was built. This metadata is primarily used to validate the use of an AST file. For example, a precompiled header built for a 32-bit x86 target cannot be used when compiling for a 64-bit x86 target. The metadata block contains information about:

Language options

Describes the particular language dialect used to compile the AST file, including major options (e.g., Objective-C support) and more minor options (e.g., support for "`//`" comments). The contents of this record correspond to the `LangOptions` class.

Target architecture

The target triple that describes the architecture, platform, and ABI for which the AST file was generated, e.g., `i386-apple-darwin9`.

AST version

The major and minor version numbers of the AST file format. Changes in the minor version number should not affect backward compatibility, while changes in the major version number imply that a newer compiler cannot read an older precompiled header (and vice-versa).

Original file name

The full path of the header that was used to generate the AST file.

Predefines buffer

Although not explicitly stored as part of the metadata, the predefines buffer is used in the validation of the AST file. The predefines buffer itself contains code generated by the compiler to initialize the preprocessor state according to the current target, platform, and command-line options. For example, the predefines buffer will contain "`#define __STDC__ 1`" when we are compiling C without Microsoft extensions. The predefines buffer itself is stored within the Source Manager Block, but its contents are verified along with the rest of the metadata.

A chained PCH file (that is, one that references another PCH) and a module (which may import other modules) have additional metadata containing the list of all AST files that this AST file depends on. Each of those files will be loaded along with this AST file.

For chained precompiled headers, the language options, target architecture and predefines buffer data is taken from the end of the chain, since they have to match anyway.

Source Manager Block

The source manager block contains the serialized representation of Clang's SourceManager class, which handles the mapping from source locations (as represented in Clang's abstract syntax tree) into actual column/line positions within a source file or macro instantiation. The AST file's representation of the source manager also includes information about all of the headers that were (transitively) included when building the AST file.

The bulk of the source manager block is dedicated to information about the various files, buffers, and macro instantiations into which a source location can refer. Each of these is referenced by a numeric "file ID", which is a unique number (allocated starting at 1) stored in the source location. Clang serializes the information for each kind of file ID, along with an index that maps file IDs to the position within the AST file where the information about that file ID is stored. The data associated with a file ID is loaded only when required by the front end, e.g., to emit a diagnostic that includes a macro instantiation history inside the header itself.

The source manager block also contains information about all of the headers that were included when building the AST file. This includes information about the controlling macro for the header (e.g., when the preprocessor identified that the contents of the header dependent on a macro like `LLVM_CLANG_SOURCEMANAGER_H`).

Preprocessor Block

The preprocessor block contains the serialized representation of the preprocessor. Specifically, it contains all of the macros that have been defined by the end of the header used to build the AST file, along with the token sequences that comprise each macro. The macro definitions are only read from the AST file when the name of the macro first occurs in the program. This lazy loading of macro definitions is triggered by lookups into the identifier table.

Types Block

The types block contains the serialized representation of all of the types referenced in the translation unit. Each Clang type node (`PointerType`, `FunctionProtoType`, etc.) has a corresponding record type in the AST file. When types are deserialized from the AST file, the data within the record is used to reconstruct the appropriate type node using the AST context.

Each type has a unique type ID, which is an integer that uniquely identifies that type. Type ID 0 represents the NULL type, type IDs less than `NUM_PREDEF_TYPE_IDS` represent predefined types (`void`, `float`, etc.), while other "user-defined" type IDs are assigned consecutively from `NUM_PREDEF_TYPE_IDS` upward as the types are encountered. The AST file has an associated mapping from the user-defined types block to the location within the types block where the serialized representation of that type resides, enabling lazy deserialization of types. When a type is referenced from within the AST file, that reference is encoded using the type ID shifted left by 3 bits. The lower three bits are used to represent the `const`, `volatile`, and `restrict` qualifiers, as in Clang's QualType class.

Declarations Block

The declarations block contains the serialized representation of all of the declarations referenced in the translation unit. Each Clang declaration node (`VarDecl`, `FunctionDecl`, etc.) has a corresponding record type in the AST file. When declarations are deserialized from the AST file, the data within the record is used to build and populate a new instance of the corresponding `Decl` node. As with types, each declaration node has a numeric ID that is used to refer to that declaration within the AST file. In addition, a lookup table provides a mapping from that numeric ID to the offset within the precompiled header where that declaration is described.

Declarations in Clang's abstract syntax trees are stored hierarchically. At the top of the hierarchy is the translation unit (`TranslationUnitDecl`), which contains all of the declarations in the translation unit but is not actually written as a specific declaration node. Its child declarations (such as functions or struct types) may also contain other declarations inside them, and so on. Within Clang, each declaration is stored within a declaration context, as represented by the `DeclContext` class. Declaration contexts provide the mechanism to perform name lookup within a given declaration (e.g., find the member named `x` in a structure) and iterate over the declarations stored within a context (e.g., iterate over all of the fields of a structure for structure layout).

In Clang's AST file format, deserializing a declaration that is a `DeclContext` is a separate operation from deserializing all of the declarations stored within that declaration context. Therefore, Clang will deserialize the translation unit declaration without deserializing the declarations within that translation unit. When required, the declarations stored within a declaration context will be deserialized. There are two representations of the declarations within a declaration context, which correspond to the name-lookup and iteration behavior described above:

- When the front end performs name lookup to find a name `x` within a given declaration context (for example, during semantic analysis of the expression `p->x`, where `p`'s type is defined in the precompiled header), Clang refers to an on-disk hash table that maps from the names within that declaration context to the declaration IDs that represent each visible declaration with that name. The actual declarations will then be deserialized to provide the results of name lookup.
- When the front end performs iteration over all of the declarations within a declaration context, all of those declarations are immediately de-serialized. For large declaration contexts (e.g., the translation unit), this operation is expensive; however, large declaration contexts are not traversed in normal compilation, since such a traversal is unnecessary. However, it is common for the code generator and semantic analysis to traverse declaration contexts for structs, classes, unions, and enumerations, although those contexts contain relatively few declarations in the common case.

Statements and Expressions

Statements and expressions are stored in the AST file in both the types and the declarations blocks, because every statement or expression will be associated with either a type or declaration. The actual statement and expression records are stored immediately following the declaration or type that owns the statement or expression. For example, the statement representing the body of a function will be stored directly following the declaration of the function.

As with types and declarations, each statement and expression kind in Clang's abstract syntax tree (`ForStmt`, `CallExpr`, etc.) has a corresponding record type in the AST file, which contains the serialized representation of that statement or expression. Each substatement or subexpression within an expression is stored as a separate record (which keeps most records to a fixed size). Within the AST file, the subexpressions of an expression are stored, in reverse order, prior to the expression that owns those expression, using a form of Reverse Polish Notation. For example, an expression `3 - 4 + 5` would be represented as follows:

```
IntegerLiteral(5)
IntegerLiteral(4)
IntegerLiteral(3)
IntegerLiteral(-)
IntegerLiteral(+)
STOP
```

When reading this representation, Clang evaluates each expression record it encounters, builds the appropriate abstract syntax tree node, and then pushes that expression on to a stack. When a record contains *N* subexpressions — `BinaryOperator` has two of them — those expressions are popped from the top of the stack. The special STOP code indicates that we have reached the end of a serialized expression or statement; other expression or statement records may follow, but they are part of a different expression.

Identifier Table Block

The identifier table block contains an on-disk hash table that maps each identifier mentioned within the AST file to the serialized representation of the identifier's information (e.g, the `IdentifierInfo` structure). The serialized representation contains:

- The actual identifier string.
- Flags that describe whether this identifier is the name of a built-in, a poisoned identifier, an extension token, or a macro.
- If the identifier names a macro, the offset of the macro definition within the Preprocessor Block.
- If the identifier names one or more declarations visible from translation unit scope, the declaration IDs of these declarations.

When an AST file is loaded, the AST file reader mechanism introduces itself into the identifier table as an external lookup source. Thus, when the user program refers to an identifier that has not yet been seen, Clang will perform a lookup into the identifier table. If an identifier is found, its contents (macro definitions, flags, top-level declarations, etc.) will be deserialized, at which point the corresponding `IdentifierInfo` structure will have the same contents it would have after parsing the headers in the AST file.

Within the AST file, the identifiers used to name declarations are represented with an integral value. A separate table provides a mapping from this integral value (the identifier ID) to the location within the on-disk hash table where that identifier is stored. This mapping is used when deserializing the name of a declaration, the identifier of a token, or any other construct in the AST file that refers to a name.

Method Pool Block

The method pool block is represented as an on-disk hash table that serves two purposes: it provides a mapping from the names of Objective-C selectors to the set of Objective-C instance and class methods that have that particular selector (which is required for semantic analysis in Objective-C) and also stores all of the selectors used by entities within the AST file. The design of the method pool is similar to that of the identifier table: the first time a particular selector is formed during the compilation of the program, Clang will search in the on-disk hash table of selectors; if found, Clang will read the Objective-C methods associated with that selector into the appropriate front-end data structure (`Sema::InstanceMethodPool` and `Sema::FactoryMethodPool` for instance and class methods, respectively).

As with identifiers, selectors are represented by numeric values within the AST file. A separate index maps these numeric selector values to the offset of the selector within the on-disk hash table, and will be used when de-serializing an Objective-C method declaration (or other Objective-C construct) that refers to the selector.

AST Reader Integration Points

The “lazy” deserialization behavior of AST files requires their integration into several completely different submodules of Clang. For example, lazily deserializing the declarations during name lookup requires that the name-lookup routines be able to query the AST file to find entities stored there.

For each Clang data structure that requires direct interaction with the AST reader logic, there is an abstract class that provides the interface between the two modules. The `ASTReader` class, which handles the loading of an AST file, inherits from all of these abstract classes to provide lazy deserialization of Clang’s data structures. `ASTReader` implements the following abstract classes:

`ExternalSLocEntrySource`

This abstract interface is associated with the `SourceManager` class, and is used whenever the source manager needs to load the details of a file, buffer, or macro instantiation.

`IdentifierInfoLookup`

This abstract interface is associated with the `IdentifierTable` class, and is used whenever the program source refers to an identifier that has not yet been seen. In this case, the AST reader searches for this identifier within its identifier table to load any top-level declarations or macros associated with that identifier.

`ExternalASTSource`

This abstract interface is associated with the `ASTContext` class, and is used whenever the abstract syntax tree nodes need to loaded from the AST file. It provides the ability to de-serialize declarations and types identified by their numeric values, read the bodies of functions when required, and read the declarations stored within a declaration context (either for iteration or for name lookup).

`ExternalSemaSource`

This abstract interface is associated with the `Sema` class, and is used whenever semantic analysis needs to read information from the global method pool.

Chained precompiled headers

Chained precompiled headers were initially intended to improve the performance of IDE-centric operations such as syntax highlighting and code completion while a particular source file is being edited by the user. To minimize the amount of reparsing required after a change to the file, a form of precompiled header — called a precompiled *preamble* — is automatically generated by parsing all of the headers in the source file, up to and including the last `#include`. When only the source file changes (and none of the headers it depends on), reparsing of that source file can use the precompiled preamble and start parsing after the `#includes`, so parsing time is proportional to the size of the source file (rather than all of its includes). However, the compilation of that translation unit may already use a precompiled header: in this case, Clang will create the precompiled preamble as a chained precompiled header that refers to the original precompiled header. This drastically reduces the time needed to serialize the precompiled preamble for use in reparsing.

Chained precompiled headers get their name because each precompiled header can depend on one other precompiled header, forming a chain of dependencies. A translation unit will then include the precompiled header that starts the chain (i.e., nothing depends on it). This linearity of dependencies is important for the semantic model of chained precompiled headers, because the most-recent precompiled header can provide information that overrides the information provided by the precompiled headers it depends on, just like a header file `B.h` that includes another header `A.h` can modify the state produced by parsing `A.h`, e.g., by `#undef`'ing a macro defined in `A.h`.

There are several ways in which chained precompiled headers generalize the AST file model:

Numbering of IDs

Many different kinds of entities — identifiers, declarations, types, etc. — have ID numbers that start at 1 or some other predefined constant and grow upward. Each precompiled header records the maximum ID number it has assigned in each category. Then, when a new precompiled header is generated that depends on (chains to) another precompiled header, it will start counting at the next available ID number. This way, one can determine, given an ID number, which AST file actually contains the entity.

Name lookup

When writing a chained precompiled header, Clang attempts to write only information that has changed from the precompiled header on which it is based. This changes the lookup algorithm for the various tables, such as the identifier table: the search starts at the most-recent precompiled header. If no entry is found, lookup then proceeds to the identifier table in the precompiled header it depends on, and so one. Once a lookup succeeds, that result is considered definitive, overriding any results from earlier precompiled headers.

Update records

There are various ways in which a later precompiled header can modify the entities described in an earlier precompiled header. For example, later precompiled headers can add entries into the various name-lookup tables for the translation unit or namespaces, or add new categories to an Objective-C class. Each of these updates is captured in an "update record" that is stored in the chained precompiled header file and will be loaded along with the original entity.

Modules

Modules generalize the chained precompiled header model yet further, from a linear chain of precompiled headers to an arbitrary directed acyclic graph (DAG) of AST files. All of the same techniques used to make chained precompiled headers work — ID number, name lookup, update records — are shared with modules. However, the DAG nature of modules introduce a number of additional complications to the model:

Numbering of IDs

The simple, linear numbering scheme used in chained precompiled headers falls apart with the module DAG, because different modules may end up with different numbering schemes for entities they imported from common shared modules. To account for this, each module file provides information about which modules it depends on and which ID numbers it assigned to the entities in those modules, as well as which ID numbers it took for its own new entities. The AST reader then maps these "local" ID numbers into a "global" ID number space for the current translation unit, providing a 1-1 mapping between entities (in whatever AST file they inhabit) and global ID numbers. If that translation unit is then serialized into an AST file, this mapping will be stored for use when the AST file is imported.

Declaration merging

It is possible for a given entity (from the language's perspective) to be declared multiple times in different places. For example, two different headers can have the declaration of `printf` or could forward-declare

`struct stat`. If each of those headers is included in a module, and some third party imports both of those modules, there is a potentially serious problem: name lookup for `printf` or `struct stat` will find both declarations, but the AST nodes are unrelated. This would result in a compilation error, due to an ambiguity in name lookup. Therefore, the AST reader performs declaration merging according to the appropriate language semantics, ensuring that the two disjoint declarations are merged into a single redeclaration chain (with a common canonical declaration), so that it is as if one of the headers had been included before the other.

Name Visibility

Modules allow certain names that occur during module creation to be “hidden”, so that they are not part of the public interface of the module and are not visible to its clients. The AST reader maintains a “visible” bit on various AST nodes (declarations, macros, etc.) to indicate whether that particular AST node is currently visible; the various name lookup mechanisms in Clang inspect the visible bit to determine whether that entity, which is still in the AST (because other, visible AST nodes may depend on it), can actually be found by name lookup. When a new (sub)module is imported, it may make existing, non-visible, already-deserialized AST nodes visible; it is the responsibility of the AST reader to find and update these AST nodes when it is notified of the import.

ABI tags

Introduction

This text tries to describe gcc semantic for mangling “abi_tag” attributes described in https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Attributes.html

There is no guarantee the following rules are correct, complete or make sense in any way as they were determined empirically by experiments with gcc5.

Declaration

ABI tags are declared in an abi_tag attribute and can be applied to a function, variable, class or inline namespace declaration. The attribute takes one or more strings (called tags); the order does not matter.

See https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Attributes.html for details.

Tags on an inline namespace are called “implicit tags”, all other tags are “explicit tags”.

Mangling

All tags that are “active” on an <unqualified-name> are emitted after the <unqualified-name>, before <template-args> or <discriminator>, and are part of the same <substitution> the <unqualified-name> is.

They are mangled as:

```
<abi-tags> ::= <abi-tag>*   # sort by name
<abi-tag> ::= B <tag source-name>
```

Example:

```
__attribute__((abi_tag("test")))
void Func();
// gets mangled as: _Z4FuncB4testv (prettified as `Func[abi:test]()`)
```

Active tags

A namespace does not have any active tags. For types (class / struct / union / enum), the explicit tags are the active tags.

For variables and functions, the active tags are the explicit tags plus any “required tags” which are not in the “available tags” set:

```
derived-tags := (required-tags - available-tags)
active-tags := explicit-tags + derived-tags
```

Required tags for a function

If a function is used as a local scope for another name, and is part of another function as local scope, it doesn't have any required tags.

If a function is used as a local scope for a guard variable name, it doesn't have any required tags.

Otherwise the function requires any implicit or explicit tag used in the name for the return type.

Example:

```
namespace A {
  inline namespace B __attribute__((abi_tag)) {
    struct C { int x; };
  }
}

A::C foo(); // gets mangled as: _Z3fooB1Bv (prettified as `foo[abi:B]()`)
```

Required tags for a variable

A variable requires any implicit or explicit tag used in its type.

Available tags

All tags used in the prefix and in the template arguments for a name are available. Also, for functions, all tags from the <bare-function-type> (which might include the return type for template functions) are available.

For <local-name>s all active tags used in the local part (<function- encoding>) are available, but not implicit tags which were not active.

Implicit and explicit tags used in the <unqualified-name> for a function (as in the type of a cast operator) are NOT available.

Example: a cast operator to std::string (which is std::__cxx11::basic_string<…>) will use 'cxx11' as an active tag, as it is required from the return type *std::string* but not available.

Hardware-assisted AddressSanitizer Design Documentation

This page is a design document for **hardware-assisted AddressSanitizer** (or **HWASAN**) a tool similar to AddressSanitizer, but based on partial hardware assistance.

Introduction

AddressSanitizer tags every 8 bytes of the application memory with a 1 byte tag (using *shadow memory*), uses *redzones* to find buffer-overflows and *quarantine* to find use-after-free. The redzones, the quarantine, and, to a less extent, the shadow, are the sources of AddressSanitizer's memory overhead. See the AddressSanitizer paper for details.

AArch64 has Address Tagging (or top-byte-ignore, TBI), a hardware feature that allows software to use the 8 most significant bits of a 64-bit pointer as a tag. HWASAN uses Address Tagging to implement a memory safety tool, similar to AddressSanitizer, but with smaller memory overhead and slightly different (mostly better) accuracy guarantees.

Intel's Linear Address Masking (LAM) also provides address tagging for x86_64, though it is not widely available in hardware yet. For x86_64, HWASAN has a limited implementation using page aliasing instead.

Algorithm

- Every heap/stack/global memory object is forcibly aligned by *TG* bytes (*TG* is e.g. 16 or 64). We call *TG* the **tagging granularity**.
- For every such object a random *TS*-bit tag *T* is chosen (*TS*, or tag size, is e.g. 4 or 8)

- The pointer to the object is tagged with *T*.
- The memory for the object is also tagged with *T* (using a *TG=>1* shadow memory)
- Every load and store is instrumented to read the memory tag and compare it with the pointer tag, exception is raised on tag mismatch.

For a more detailed discussion of this approach see https://arxiv.org/pdf/1802.09517.pdf

Short granules

A short granule is a granule of size between 1 and *TG-1* bytes. The size of a short granule is stored at the location in shadow memory where the granule's tag is normally stored, while the granule's actual tag is stored in the last byte of the granule. This means that in order to verify that a pointer tag matches a memory tag, HWASAN must check for two possibilities:

- the pointer tag is equal to the memory tag in shadow memory, or
- the shadow memory tag is actually a short granule size, the value being loaded is in bounds of the granule and the pointer tag is equal to the last byte of the granule.

Pointer tags between 1 to *TG-1* are possible and are as likely as any other tag. This means that these tags in memory have two interpretations: the full tag interpretation (where the pointer tag is between 1 and *TG-1* and the last byte of the granule is ordinary data) and the short tag interpretation (where the pointer tag is stored in the granule).

When HWASAN detects an error near a memory tag between 1 and *TG-1*, it will show both the memory tag and the last byte of the granule. Currently, it is up to the user to disambiguate the two possibilities.

Instrumentation

Memory Accesses

In the majority of cases, memory accesses are prefixed with a call to an outlined instruction sequence that verifies the tags. The code size and performance overhead of the call is reduced by using a custom calling convention that

- preserves most registers, and
- is specialized to the register containing the address, and the type and size of the memory access.

Currently, the following sequence is used:

```
// int foo(int *a) { return *a; }
// clang -O2 --target=aarch64-linux-android30 -fsanitize=hwaddress -S -o - load.c
[...]
foo:
      stp     x30, x20, [sp, #-16]!
      adrp    x20, :got:__hwasan_shadow               // load shadow address from GOT into x20
      ldr     x20, [x20, :got_lo12:__hwasan_shadow]
      bl      __hwasan_check_x0_2_short_v2            // call outlined tag check
                                                      // (arguments: x0 = address, x20 = shadow base;
                                                      // "2" encodes the access type and size)
      ldr     w0, [x0]                                // inline load
      ldp     x30, x20, [sp], #16
      ret

[...]
__hwasan_check_x0_2_short_v2:
      sbfx    x16, x0, #4, #52                        // shadow offset
      ldrb    w16, [x20, x16]                         // load shadow tag
      cmp     x16, x0, lsr #56                        // extract address tag, compare with shadow tag
      b.ne    .Ltmp0                                  // jump to short tag handler on mismatch
.Ltmp1:
      ret
.Ltmp0:
      cmp     w16, #15                                // is this a short tag?
      b.hi    .Ltmp2                                  // if not, error
      and     x17, x0, #0xf                           // find the address's position in the short granule
      add     x17, x17, #3                            // adjust to the position of the last byte loaded
      cmp     w16, w17                                // check that position is in bounds
      b.ls    .Ltmp2                                  // if not, error
      orr     x16, x0, #0xf                           // compute address of last byte of granule
      ldrb    w16, [x16]                              // load tag from it
      cmp     x16, x0, lsr #56                        // compare with pointer tag
      b.eq    .Ltmp1                                  // if matches, continue
.Ltmp2:
      stp     x0, x1, [sp, #-256]!                    // save original x0, x1 on stack (they will be overwritten)
      stp     x29, x30, [sp, #232]                    // create frame record
      mov     x1, #2                                  // set x1 to a constant indicating the type of failure
      adrp    x16, :got:__hwasan_tag_mismatch_v2      // call runtime function to save remaining registers and report error
      ldr     x16, [x16, :got_lo12:__hwasan_tag_mismatch_v2] // (load address from GOT to avoid potential register clobbers in delay load handler)
      br      x16
```

Heap

Tagging the heap memory/pointers is done by *malloc*. This can be based on any malloc that forces all objects to be TG-aligned. *free* tags the memory with a different tag.

Stack

Stack frames are instrumented by aligning all non-promotable allocas by *TG* and tagging stack memory in function prologue and epilogue.

Tags for different allocas in one function are **not** generated independently; doing that in a function with *M* allocas would require maintaining *M* live stack pointers, significantly increasing register pressure. Instead we generate a single base tag value in the prologue, and build the tag for alloca number *M* as *ReTag(BaseTag, M)*, where ReTag can be as simple as exclusive-or with constant *M*.

Stack instrumentation is expected to be a major source of overhead, but could be optional.

Globals

Most globals in HWASAN instrumented code are tagged. This is accomplished using the following mechanisms:

- The address of each global has a static tag associated with it. The first defined global in a translation unit has a pseudorandom tag associated with it, based on the hash of the file path. Subsequent global tags are incremental from the previously-assigned tag.
- The global's tag is added to its symbol address in the object file's symbol table. This causes the global's address to be tagged when its address is taken.
- When the address of a global is taken directly (i.e. not via the GOT), a special instruction sequence needs to be used to add the tag to the address, because the tag would otherwise take the address outside of the small code model (4GB on AArch64). No changes are required when the address is taken via the GOT because the address stored in the GOT will contain the tag.
- An associated `hwasan_globals` section is emitted for each tagged global, which indicates the address of the global, its size and its tag. These sections are concatenated by the linker into a single `hwasan_globals` section that is enumerated by the runtime (via an ELF note) when a binary is loaded and the memory is tagged accordingly.

A complete example is given below:

```
// int x = 1; int *f() { return &x; }
// clang -O2 --target=aarch64-linux-android30 -fsanitize=hwaddress -S -o - global.c

[...]
f:
      adrp    x0, :pg_hi21_nc:x           // set bits 12-63 to upper bits of untagged address
      movk    x0, #:prel_g3:x+0x100000000  // set bits 48-63 to tag
      add     x0, x0, :lo12:x             // set bits 0-11 to lower bits of address
      ret

[...]
      .data
.Lx.hwasan:
      .word   1

      .globl  x
      .set x, .Lx.hwasan+0x2d00000000000000

[...]
      .section        .note.hwasan.globals,"aG",@note,hwasan.module_ctor,comdat
.Lhwasan.note:
      .word   8                           // namesz
      .word   8                           // descsz
      .word   3                           // NT_LLVM_HWASAN_GLOBALS
```

```
        .asciz  "LLVM\000\000\000"
        .word   __start_hwasan_globals-.Lhwasan.note
        .word   __stop_hwasan_globals-.Lhwasan.note

[...]
        .section        hwasan_globals,"ao",@progbits,.Lx.hwasan,unique,2
.Lx.hwasan.descriptor:
        .word   .Lx.hwasan-.Lx.hwasan.descriptor
        .word   0x2d000004                  // tag = 0x2d, size = 4
```

Error reporting

Errors are generated by the *HLT* instruction and are handled by a signal handler.

Attribute

HWASAN uses its own LLVM IR Attribute *sanitize_hwaddress* and a matching C function attribute. An alternative would be to re-use ASAN's attribute *sanitize_address*. The reasons to use a separate attribute are:

- Users may need to disable ASAN but not HWASAN, or vise versa, because the tools have different trade-offs and compatibility issues.
- LLVM (ideally) does not use flags to decide which pass is being used, ASAN or HWASAN are being applied, based on the function attributes.

This does mean that users of HWASAN may need to add the new attribute to the code that already uses the old attribute.

Comparison with AddressSanitizer

HWASAN:

- Is less portable than AddressSanitizer as it relies on hardware Address Tagging (AArch64). Address Tagging can be emulated with compiler instrumentation, but it will require the instrumentation to remove the tags before any load or store, which is infeasible in any realistic environment that contains non-instrumented code.
- May have compatibility problems if the target code uses higher pointer bits for other purposes.
- May require changes in the OS kernels (e.g. Linux seems to dislike tagged pointers passed from address space: https://www.kernel.org/doc/Documentation/arm64/tagged-pointers.txt).
- **Does not require redzones to detect buffer overflows**, but the buffer overflow detection is probabilistic, with roughly *1/(2**TS)* chance of missing a bug (6.25% or 0.39% with 4 and 8-bit TS respectively).
- **Does not require quarantine to detect heap-use-after-free, or stack-use-after-return**. The detection is similarly probabilistic.

The memory overhead of HWASAN is expected to be much smaller than that of AddressSanitizer: *1/TG* extra memory for the shadow and some overhead due to *TG*-aligning all objects.

Supported architectures

HWASAN relies on Address Tagging which is only available on AArch64. For other 64-bit architectures it is possible to remove the address tags before every load and store by compiler instrumentation, but this variant will have limited deployability since not all of the code is typically instrumented.

On x86_64, HWASAN utilizes page aliasing to place tags in userspace address bits. Currently only heap tagging is supported. The page aliases rely on shared memory, which will cause heap memory to be shared between processes if the application calls `fork()`. Therefore x86_64 is really only safe for applications that do not fork.

HWASAN does not currently support 32-bit architectures since they do not support Address Tagging and the address space is too constrained to easily implement page aliasing.

Related Work

- SPARC ADI implements a similar tool mostly in hardware.
- Effective and Efficient Memory Protection Using Dynamic Tainting discusses similar approaches (“lock & key”).
- Watchdog discussed a heavier, but still somewhat similar “lock & key” approach.
- *TODO: add more “related work” links. Suggestions are welcome.*

Constant Interpreter

Introduction **990**
Bytecode Compilation **990**
Primitive Types 990
Composite types 991
Bytecode Execution **991**
Memory Organisation **991**
Blocks 991
Descriptors 992
Pointers 993
BlockPointer 993
ExternPointer 994
TargetPointer 994
TypeInfoPointer 994
InvalidPointer 994
TODO **994**
Missing Language Features 994
Known Bugs 995

Introduction

The constexpr interpreter aims to replace the existing tree evaluator in clang, improving performance on constructs which are executed inefficiently by the evaluator. The interpreter is activated using the following flags:

- `-fexperimental-new-constant-interpreter` enables the interpreter, emitting an error if an unsupported feature is encountered

Bytecode Compilation

Bytecode compilation is handled in `ByteCodeStmtGen.h` for statements and `ByteCodeExprGen.h` for expressions. The compiler has two different backends: one to generate bytecode for functions (`ByteCodeEmitter`) and one to directly evaluate expressions as they are compiled, without generating bytecode (`EvalEmitter`). All functions are compiled to bytecode, while toplevel expressions used in constant contexts are directly evaluated since the bytecode would never be reused. This mechanism aims to pave the way towards replacing the evaluator, improving its performance on functions and loops, while being just as fast on single-use toplevel expressions.

The interpreter relies on stack-based, strongly-typed opcodes. The glue logic between the code generator, along with the enumeration and description of opcodes, can be found in `Opcodes.td`. The opcodes are implemented as generic template methods in `Interp.h` and instantiated with the relevant primitive types by the interpreter loop or by the evaluating emitter.

Primitive Types

- `PT_{U|S}int{8|16|32|64}`

Signed or unsigned integers of a specific bit width, implemented using the `Integral` type.

- `PT_{U|S}intFP`

  Signed or unsigned integers of an arbitrary, but fixed width used to implement integral types which are required by the target, but are not supported by the host. Under the hood, they rely on APValue. The `Integral` specialisation for these types is required by opcodes to share an implementation with fixed integrals.

- `PT_Bool`

  Representation for boolean types, essentially a 1-bit unsigned `Integral`.

- `PT_RealFP`

  Arbitrary, but fixed precision floating point numbers. Could be specialised in the future similarly to integers in order to improve floating point performance.

- `PT_Ptr`

  Pointer type, defined in `"Pointer.h"`. A pointer can be either null, reference interpreter-allocated memory (`BlockPointer`) or point to an address which can be derived, but not accessed (`ExternPointer`).

- `PT_FnPtr`

  Function pointer type, can also be a null function pointer. Defined in `"FnPointer.h"`.

- `PT_MemPtr`

  Member pointer type, can also be a null member pointer. Defined in `"MemberPointer.h"`

- `PT_VoidPtr`

  Void pointer type, can be used for rount-trip casts. Represented as the union of all pointers which can be cast to void. Defined in `"VoidPointer.h"`.

- `PT_ObjCBlockPtr`

  Pointer type for ObjC blocks. Defined in `"ObjCBlockPointer.h"`.

Composite types

The interpreter distinguishes two kinds of composite types: arrays and records (structs and classes). Unions are represented as records, except at most a single field can be marked as active. The contents of inactive fields are kept until they are reactivated and overwritten. Complex numbers (`_Complex`) and vectors (`__attribute((vector_size(16)))`) are treated as arrays.

Bytecode Execution

Bytecode is executed using a stack-based interpreter. The execution context consists of an `InterpStack`, along with a chain of `InterpFrame` objects storing the call frames. Frames are built by call instructions and destroyed by return instructions. They perform one allocation to reserve space for all locals in a single block. These objects store all the required information to emit stack traces whenever evaluation fails.

Memory Organisation

Memory management in the interpreter relies on 3 data structures: `Block` objects which store the data and associated inline metadata, `Pointer` objects which refer to or into blocks, and `Descriptor` structures which describe blocks and subobjects nested inside blocks.

Blocks

Blocks contain data interleaved with metadata. They are allocated either statically in the code generator (globals, static members, dummy parameter values etc.) or dynamically in the interpreter, when creating the frame containing the local variables of a function. Blocks are associated with a descriptor that characterises the entire allocation, along with a few additional attributes:

- `IsStatic` indicates whether the block has static duration in the interpreter, i.e. it is not a local in a frame.

- `DeclID` identifies each global declaration (it is set to an invalid and irrelevant value for locals) in order to prevent illegal writes and reads involving globals and temporaries with static storage duration.

Static blocks are never deallocated, but local ones might be deallocated even when there are live pointers to them. Pointers are only valid as long as the blocks they point to are valid, so a block with pointers to it whose lifetime ends is kept alive until all pointers to it go out of scope. Since the frame is destroyed on function exit, such blocks are turned into a `DeadBlock` and copied to storage managed by the interpreter itself, not the frame. Reads and writes to these blocks are illegal and cause an appropriate diagnostic to be emitted. When the last pointer goes out of scope, dead blocks are also deallocated.

The lifetime of blocks is managed through 3 methods stored in the descriptor of the block:

- **CtorFn**: initializes the metadata which is store in the block, alongside actual data. Invokes the default constructors of objects which are not trivial (`Pointer`, `RealFP`, etc.)
- **DtorFn**: invokes the destructors of non-trivial objects.
- **MoveFn**: moves a block to dead storage.

Non-static blocks track all the pointers into them through an intrusive doubly-linked list, required to adjust and invalidate all pointers when transforming a block into a dead block. If the lifetime of an object ends, all pointers to it are invalidated, emitting the appropriate diagnostics when dereferenced.

The interpreter distinguishes 3 different kinds of blocks:

- **Primitives**

  A block containing a single primitive with no additional metadata.

- **Arrays of primitives**

  An array of primitives contains a pointer to an `InitMap` storage as its first field: the initialisation map is a bit map indicating all elements of the array which were initialised. If the pointer is null, no elements were initialised, while a value of `(InitMap*)-1` indicates that the object was fully initialised. When all fields are initialised, the map is deallocated and replaced with that token.

  Array elements are stored sequentially, without padding, after the pointer to the map.

- **Arrays of composites and records**

  Each element in an array of composites is preceded by an `InlineDescriptor` which stores the attributes specific to the field and not the whole allocation site. Descriptors and elements are stored sequentially in the block. Records are laid out identically to arrays of composites: each field and base class is preceded by an inline descriptor. The `InlineDescriptor` has the following fields:

  - **Offset**: byte offset into the array or record, used to step back to the parent array or record.
  - **IsConst**: flag indicating if the field is const-qualified.
  - **IsInitialized**: flag indicating whether the field or element was initialized. For non-primitive fields, this is only relevant to determine the dynamic type of objects during construction.
  - **IsBase**: flag indicating whether the record is a base class. In that case, the offset can be used to identify the derived class.
  - **IsActive**: indicates if the field is the active field of a union.
  - **IsMutable**: indicates if the field is marked as mutable.

Inline descriptors are filled in by the *CtorFn* of blocks, which leaves storage in an uninitialised, but valid state.

Descriptors

Descriptors are generated at bytecode compilation time and contain information required to determine if a particular memory access is allowed in constexpr. They also carry all the information required to emit a diagnostic involving a memory access, such as the declaration which originates the block. Currently there is a single kind of descriptor encoding information for all block types.

Pointers

Pointers, implemented in `Pointer.h` are represented as a tagged union. Some of these may not yet be available in upstream `clang`.

- **BlockPointer**: used to reference memory allocated and managed by the interpreter, being the only pointer kind which allows dereferencing in the interpreter
- **ExternPointer**: points to memory which can be addressed, but not read by the interpreter. It is equivalent to APValue, tracking a declaration and a path of fields and indices into that allocation.
- **TargetPointer**: represents a target address derived from a base address through pointer arithmetic, such as `((int *)0x100)[20]`. Null pointers are target pointers with a zero offset.
- **TypeInfoPointer**: tracks information for the opaque type returned by `typeid`
- **InvalidPointer**: is dummy pointer created by an invalid operation which allows the interpreter to continue execution. Does not allow pointer arithmetic or dereferencing.

Besides the previously mentioned union, a number of other pointer-like types have their own type:

- **ObjCBlockPointer** tracks Objective-C blocks
- **FnPointer** tracks functions and lazily caches their compiled version
- **MemberPointer** tracks C++ object members

Void pointers, which can be built by casting any of the aforementioned pointers, are implemented as a union of all pointer types. The `BitCast` opcode is responsible for performing all legal conversions between these types and primitive integers.

BlockPointer

Block pointers track a `Pointee`, the block to which they point, along with a `Base` and an `Offset`. The base identifies the innermost field, while the offset points to an array element relative to the base (including one-past-end pointers). The offset identifies the array element or field which is referenced, while the base points to the outer object or array which contains the field. These two fields allow all pointers to be uniquely identified, disambiguated and characterised.

As an example, consider the following structure:

```
struct A {
    struct B {
        int x;
        int y;
    } b;
    struct C {
        int a;
        int b;
    } c[2];
    int z;
};
constexpr A a;
```

On the target, `&a` and `&a.b.x` are equal. So are `&a.c[0]` and `&a.c[0].a`. In the interpreter, all these pointers must be distinguished since the are all allowed to address distinct range of memory.

In the interpreter, the object would require 240 bytes of storage and would have its field interleaved with metadata. The pointers which can be derived to the object are illustrated in the following diagram:

```
    0   16  32  40  56  64  80  96  112 120 136 144 160 176 184 200 208 224 240
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+ B | D | D | x | D | y | D | D | D | a | D | b | D | D | a | D | b | D | z |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    ^   ^   ^       ^       ^   ^   ^       ^       ^   ^       ^       ^
    |   |   |       |       |   |   |   &a.c[0].b   |   |   &a.c[1].b   |
```

```
a    |&a.b.x    &a.y     &a.c  |&a.c[0].a          |&a.c[1].a          |
  &a.b                      &a.c[0]           &a.c[1]                &a.z
```

The `Base` offset of all pointers points to the start of a field or an array and is preceded by an inline descriptor (unless `Base` is zero, pointing to the root). All the relevant attributes can be read from either the inline descriptor or the descriptor of the block.

Array elements are identified by the `Offset` field of pointers, pointing to past the inline descriptors for composites and before the actual data in the case of primitive arrays. The `Offset` points to the offset where primitives can be read from. As an example, `a.c + 1` would have the same base as `a.c` since it is an element of `a.c`, but its offset would point to `&a.c[1]`. The array-to-pointer decay operation adjusts a pointer to an array (where the offset is equal to the base) to a pointer to the first element.

ExternPointer

Extern pointers can be derived, pointing into symbols which are not readable from constexpr. An external pointer consists of a base declaration, along with a path designating a subobject, similar to the `LValuePath` of an APValue. Extern pointers can be converted to block pointers if the underlying variable is defined after the pointer is created, as is the case in the following example:

```
extern const int a;
constexpr const int *p = &a;
const int a = 5;
static_assert(*p == 5, "x");
```

TargetPointer

While null pointer arithmetic or integer-to-pointer conversion is banned in constexpr, some expressions on target offsets must be folded, replicating the behaviour of the `offsetof` builtin. Target pointers are characterised by 3 offsets: a field offset, an array offset and a base offset, along with a descriptor specifying the type the pointer is supposed to refer to. Array indexing adjusts the array offset, while the field offset is adjusted when a pointer to a member is created. Casting an integer to a pointer sets the value of the base offset. As a special case, null pointers are target pointers with all offsets set to 0.

TypeInfoPointer

`TypeInfoPointer` tracks two types: the type assigned to `std::type_info` and the type which was passed to `typeinfo`.

InvalidPointer

Such pointers are built by operations which cannot generate valid pointers, allowing the interpreter to continue execution after emitting a warning. Inspecting such a pointer stops execution.

TODO

Missing Language Features

- Changing the active field of unions
- `volatile`
- `__builtin_constant_p`
- `dynamic_cast`
- `new` and `delete`
- Fixed Point numbers and arithmetic on Complex numbers
- Several builtin methods, including string operations and `__builtin_bit_cast`

- Continue-after-failure: a form of exception handling at the bytecode level should be implemented to allow execution to resume. As an example, argument evaluation should resume after the computation of an argument fails.
- Pointer-to-Integer conversions
- Lazy descriptors: the interpreter creates a `Record` and `Descriptor` when it encounters a type: ones which are not yet defined should be lazily created when required

Known Bugs

- If execution fails, memory storing APInts and APFloats is leaked when the stack is cleared

Indices and tables

- **genindex**
- **modindex**
- **search**