

XL C/C++ Enterprise Edition for AIX



Getting Started with XL C/C++

Version 7.0

Note!

Before using this information and the product it supports, be sure to read the information in "Notices" on page 75.

First Edition (September, 2004)

This edition applies to Version 7.0.0 of XL C/C++ Enterprise Edition for AIX® (product number 5724-I11) and to all subsequent releases and modifications until otherwise indicated in new editions.

IBM welcomes your comments. You can send them by the Internet to the following address:

`compinfo@ca.ibm.com`

Include the title and order number of this book, and the page number or topic related to your comment. Be sure to include your e-mail address if you want a reply.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 2004. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this book	v
Highlighting conventions	v
How to read the syntax diagrams	v

XL C/C++ overview	1
Command-line C and C++ compiler	1
Libraries.	1
Standard C++ library	2
IBM Mathematics Acceleration Subsystem libraries	2
IBM Distributed Debugger.	2
Other tools and utilities.	3
National language support	4
Documentation and online help	4

What's new in version 7.	7
Performance and optimization	7
Machine architecture and hardware.	7
New built-in functions for POWER5 processors.	7
New XL C/C++ pragmas	9
New optimization utilities	9
IBM Mathematics Accelerated Subsystem (MASS) libraries	10
SMP thread binding	10
Conformance to industry standards	11
Ease of use	13
New XL C/C++ options	13

Customizing the compilation environment	17
Environment variables.	17
Create symbolic links for the path.	18
Configuration files	18

Controlling the compilation process	19
Invoking the compiler	19
Object model	20
Types of input and output files.	20
Default behavior.	21

Getting started with compiler options	23
Compiler messages	23
Return codes	24
Compiler message format	24
Reusing GNU C and C++ compiler options with gxc and gxc++	25
gxc and gxc++ syntax	25
GNU C and C++ to XL C/C++ option mapping	26
Configuring the option mapping	29
Options summary: C compiler	31
Basic translation.	32
Special handling and control	33
Linking and library-related options	33

Options summary: C++ compiler	34
-----------------------------------------	----

Getting started with optimization	35
Selected compiler options for optimization	36
Getting started with optimization pragmas	38

Porting considerations	39
Portability issues intrinsic to the language	39
Diagnostics for compile-time errors	41
32- and 64-bit application development	41
32- and 64-bit development environments on AIX	43
Objects and libraries on AIX.	44
Difference between a shared object and library on AIX	45
Difference between shared and static objects on AIX	46
Link time and load time	46
Diagnostics for link-time errors.	47
Diagnostics for run-time errors	48
Shared memory parallelization	49
OpenMP directives	50
Threads on AIX	50
Features related to GNU C and C++ portability	58
Function attributes	58
Variable attributes	59
Type attributes	60
GNU C and C++ assertions	60
Other extensions related to GNU C and C++	60

Appendix A. Language support	63
Compatibility with ISO/IEC International Standards	63
ISO/IEC 14882:2003(E) International Standard compatibility	63
ISO/IEC 9899:1990 International Standard compatibility	63
ISO/IEC 9899:1999 International Standard support.	63
Enhanced language level support	66

Appendix B. OpenMP compliance and support	67
OpenMP directives	67
OpenMP data scope attribute clauses.	69
OpenMP library functions	69
OpenMP environment variables	71
OpenMP implementation-defined behavior.	72
Tuning an OpenMP program	73

Notices	75
Programming interface information	77
Trademarks and service marks	77
Industry standards	77

About this book

XL C/C++ Enterprise Edition for AIX is an optimizing, standards-based, command-line compiler for the AIX operating system running on the PowerPC architecture. The compiler is a professional programming tool for creating and maintaining 32- and 64-bit applications in the extended C and C++ programming languages.

This book introduces you to the XL C/C++ compiler. It describes the various compiler invocations and ways to customize the compilation environment and control the compilation process. This book contains descriptions of the types of transformations the compiler can perform, the accepted file types for input and output, categorized summaries of compiler options, and considerations for porting an existing application. This book also provides a brief introduction to optimizing the performance of your applications. The optimizing capabilities of the compiler enable you to exploit the multilayered architecture of the PowerPC processor.

AIX and Linux[®] are complementary operating systems. Makefiles created for applications developed with XL C/C++ can be readily adapted to be reused for porting to the Linux platform. This book is intended to help you to develop and maintain your programs with XL C/C++ and to achieve improved performance at compile, link, and run time.

This document assumes that you are familiar with the C and C++ programming languages, the AIX operating system, and the ksh shell.

Highlighting conventions

Bold	Identifies commands, keywords, and other items whose names are predefined by the system.
<i>Italics</i>	Identify parameters whose actual names or values are to be supplied by the programmer. <i>Italics</i> are also used for the first mention of new terms.
Example	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code, messages from the system, or information that you should actually type.

Examples are intended to be instructional and do not attempt to minimize run time, conserve storage, or check for errors. The examples do not demonstrate all of the possible uses of the language constructs. Some examples are only code fragments and will not compile without additional code.

How to read the syntax diagrams

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.
The ►— symbol indicates the beginning of a command, directive, or statement.
The —> symbol indicates that the command, directive, or statement syntax is continued on the next line.

Reading the Syntax Diagrams

The \blacktriangleright — symbol indicates that a command, directive, or statement is continued from the previous line.

The — \blacktriangleright symbol indicates the end of a command, directive, or statement.

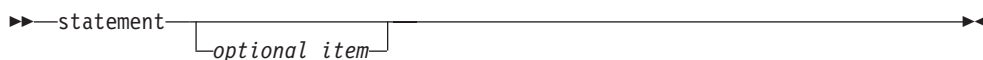
Diagrams of syntactical units other than complete commands, directives, or statements start with the \blacktriangleright — symbol and end with the — \blacktriangleright symbol.

Note: In the following diagrams, *statement* represents a C or C++ command, directive, or statement.

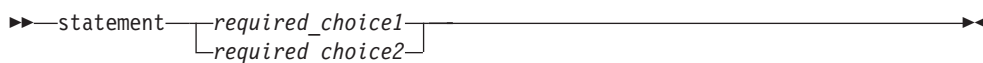
- Required items appear on the horizontal line (the main path).



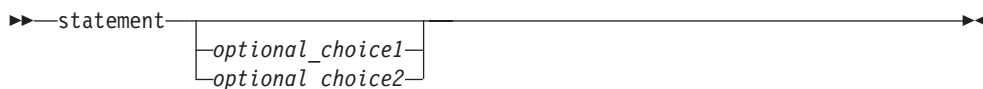
- Optional items appear below the main path.



- If you can choose from two or more items, they appear vertically, in a stack. If you *must* choose one of the items, one item of the stack appears on the main path.



If choosing one of the items is optional, the entire stack appears below the main path.



The item that is the default appears above the main path.



- An arrow returning to the left above the main line indicates an item that can be repeated.



A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

- Keywords appear in nonitalic letters and should be entered exactly as shown (for example, `extern`).

Variables appear in italicized lowercase letters (for example, *identifier*). They represent user-supplied names or values.

- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Reading the Syntax Diagrams

XL C/C++ overview

IBM XL C/C++ Enterprise Edition V7.0 is an optimizing, standards-based, command-line compiler for the AIX operating system and the PowerPC architecture. The compiler is a professional programming tool for creating and maintaining 32-bit and 64-bit applications in the extended C and C++ programming languages. The compiler represents a mature technology, which has evolved with an emphasis on performance and cross-platform portability and which contains flexibility, features, and refinements acquired from releases on other platforms.

This product is the follow-on release to IBM VisualAge C++ Professional for AIX Version 7.0. IBM has rebranded *VisualAge C++* as *XL C/C++*.

In addition to the compiler itself, XL C/C++ ships with libraries, utilities, and tools that can help you create programs efficiently. The compiler documentation includes a searchable help system, PDF books, and man pages for the compiler invocations, as well as for all command-line utilities.

Command-line C and C++ compiler

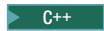
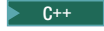
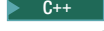
XL C/C++ provides a selection of base compiler invocation commands, which support various version levels of the C and C++ languages. Each invocation command automatically sets a compiler suboption for language level, options for other related language features, and any related predefined macros. In most cases, you should use the `xlc` command to compile C source files and the `xlc` command to compile C++ source files, or when you have both C and C++ source files.



Variations of the base command are provided to support the requirements of special environments and file systems. The variations are formed by attaching suffixes to the base command. For example, using the commands that have the suffix `_r` ensures that the compiler uses the reentrant and thread-safe versions of functions and libraries.

In addition, the `gxlc` and `gxlc++` utilities are specialized compiler invocations.

Libraries

XL C/C++ ships with the following libraries.

- SMP Runtime Library supports both explicit and automated parallel processing.
- IBM Mathematics Acceleration Subsystem (MASS) library of tuned mathematical intrinsic functions, for 32- and 64-bit modes.
- Memory Debug Runtime is used for diagnosing memory leaks.
-  Standard C++ Library (including Standard C Library and Standard Template Library) can be used to create code compliant with Standard C++.
-  C++ Runtime Library contains support routines needed by the compiler.
-  USL Complex Mathematics Class Library contains classes for manipulating complex numbers. This library is provided for use by old applications. For new applications, you should use the Standard C++ Library.

-  UNIX System Laboratories (USL) I/O Stream Class Library contains stream classes for input and output capabilities for C++. This library is provided for use by old applications. For new applications, you should use the Standard C++ Library.
-  The demangler library provides routines and classes for demangling linkage names created by the C++ compiler.

Standard C++ library

XL C/C++ Enterprise Edition for AIX ships a modified version of the Dinkum C++ Library, a conforming implementation of the Standard C++ Library. The Standard C++ Library consists of 51 headers, including 13 headers which constitute the Standard Template Library (STL). In addition, the Standard C++ Library works in conjunction with the 18 headers from the Standard C Library. The functions in these headers perform essential services such as input and output. They also provide efficient implementations of frequently used operations.

Related References

- C++ Library Overview in *Standard C++ Library Reference*

IBM Mathematics Acceleration Subsystem libraries

Starting in Version 7, XL C/C++ ships the IBM Mathematics Acceleration Subsystem (MASS) libraries of tuned mathematical intrinsic functions, for 32- and 64-bit modes. MASS libraries are thread-safe and offer improved performance over the corresponding `libm` routines. Moreover, the MASS libraries can be used without requiring code changes.

IBM Distributed Debugger

XL C/C++ ships with IBM Distributed Debugger, a client/server application that enables you to detect and diagnose errors in your programs. The client/server design makes it possible to debug both programs running on systems that are accessible through a network connection and programs that are on your workstation.

The debugger server, also known as a *debug engine*, runs on the same system where the program you want to debug runs. This system can be your workstation or a system accessible through a network. If you debug a program running on your workstation, you are performing *local debugging*. If you debug a program running on a system accessible through a network connection, you are performing *remote debugging*.

The Distributed Debugger client is a graphical user interface where you can issue commands used by the debug engine to control the execution of your program. For example, you can set breakpoints, step through your code, and examine the contents of variables. The Distributed Debugger user interface lets you debug multiple applications, which might be written in different languages, from a single debugger session. Each program you debug is shown on a separate program page. The type of information that is displayed depends on the debug engine that you are connected to.

By default, the Distributed Debugger is installed to the `/usr/idebug` directory. To start the Distributed Debugger, type `idebug` on the command line.

Other tools and utilities

CreateExportList Command

Creates a file containing a list of all the global symbols found in a given set of object files.

c++filt Name Demangling Utility

When XL C/C++ compiles a C++ program, it encodes all function names and certain other identifiers to include type and scoping information. This encoding process is called mangling. This utility converts the mangled names to their original source code names.

linkx1C Command

Links C++ .o and .a files. This command is used for linking on systems without XL C/C++ compiler installed.

makeC++SharedLib Command

Permits the creation of C++ shared libraries on systems on which the XL C/C++ compiler is not installed.

cleanpdf Command

A command related to profile-directed feedback, used for managing the PDFDIR directory. Removes all profiling information from the specified directory, the PDFDIR directory, or the current directory.

mergepdf Command

A command related to profile-directed feedback (PDF) that provides the ability to weight the importance of two or more PDF records when combining them into a single record. The PDF records must be derived from the same executable.

resetpdf Command

The current behavior of the resetpdf command is the same as the cleanpdf command and is retained for compatibility with earlier releases on other platforms.

showpdf Command

A command to display the call and block counts for all procedures executed in a profile-directed feedback training run (compilation under the options -qpdf1 and -qshowpdf).

gxc and gxc++ Utilities

Invocation methods that translate a GNU C or GNU C++ invocation command into a corresponding xlc or x1C command and invokes the XL C/C++ compiler. The purpose of these utilities is to minimize the number of changes to makefiles used for existing applications built with the GNU compilers and to facilitate the transition to XL C/C++.

Related References

- "CreateExportList Command" in *XL C/C++ Programming Guide*
- "c++filt Name Demangling Utility" in *XL C/C++ Programming Guide*
- "linkx1C Command" in *XL C/C++ Programming Guide*
- "makeC++SharedLib Command" in *XL C/C++ Programming Guide*

National language support

XL C/C++ provides support for the Unicode standard, multibyte characters, UTF-16 and UTF-32 string literals, multiple loaded locales, and bidirectionality. These features make possible or facilitate the creation of international applications.

Related References

- "The Unicode Standard" in *XL C/C++ Language Reference*
- "National Languages Support" in *XL C/C++ Compiler Reference*

Documentation and online help

XL C/C++ Enterprise Edition for AIX provides product documentation in the following formats:

- Readme files.
- Installable man pages.
- A searchable HTML-based help system.
- PDF documents.

These items are located or accessed as follows:

Readme files	The readme files are located in /usr/vacpp/ directory and in the root directory of the installation CD.
Man pages	Man pages are provided for the compiler invocations and all command-line utilities provided with the product.
HTML-based help system	A searchable help system called an Information Center, composed of HTML files. The help system is installable on a private intranet or viewable online on the product web site.
PDF documents	The PDF files are located in the /usr/vacpp/doc/\$LANG/pdf directory. They are viewable and printable from the Adobe Acrobat Reader. If you do not have the Adobe Acrobat Reader installed, you can download it from http://www.adobe.com .

The complete library of XL C/C++ PDF documents consists of the following files:

install.pdf

XL C/C++ Installation Guide contains instructions for installing the compiler, enabling the man pages, and setting up the searchable HTML help system.

getstart.pdf

Getting Started with XL C/C++ contains an overview of XL C/C++ components, explanation of new features, how-to information on customizing the compilation environment and process, summary tables of the compiler options arranged by category, an introduction to performance optimization and tuning, and general advice for porting an application to the AIX platform.

language.pdf

XL C/C++ Language Reference contains information about the IBM implementations of the C and C++ programming languages, including the implementation-defined extensions for porting an application originally developed with GNU C and g++.

compiler.pdf

XL C/C++ Compiler Reference contains information about the various compiler options, pragmas, macros, and built-in functions, including those used for parallel processing.

stdlib.pdf

Standard C++ Library Reference contains detailed information about the Standard C++ Library, including the Standard Template Library, which ship with XL C/C++.

proguide.pdf

XL C/C++ Programming Guide contains information about programming using XL C/C++ not covered in other publications.

legacy.pdf

C/C++ Legacy Class Libraries Reference contains information about the USL Complex Mathematics Class Library and the USL I/O Stream Library, which ship with XL C/C++ Enterprise Edition for compatibility with previous releases of the compiler.

debug.pdf

IBM Distributed Debugger contains the documentation for the Distributed Debugger tool.

Accessing additional information

For the latest information about XL C/C++, visit the product documentation and support pages at the following URLs. In addition, IBM Redbooks, developed by the IBM Technical Support Organization, contain technical information based on realistic scenarios from practical experience.

- The Information Center at <http://www.ibm.com/software/awdtools/vacpp/library>.
- The product support site at <http://www.ibm.com/software/awdtools/ccompilers>.
- IBM Redbooks at <http://www.redbooks.ibm.com>.

You might find the following Redbooks useful for application development with XL C/C++:

- *AIX 5L Porting Guide*, SG24-6034-00.
- *Developing and Porting C and C++ Applications on AIX*, SG24-5674-01.
- *POWER4 Processor Introduction and Tuning Guide*, SG24-7041-00.
- *Scientific Applications in RS/6000 SP Environments*, SG24-5611-00.
- *Understanding IBM eServer pSeries Performance and Sizing*, SG24-4810-01.

What's new in version 7

The new features and enhancements in XL C/C++ Enterprise Edition for AIX fall into three categories: performance and optimization, conformance to industry standards, and ease of use.

Performance and optimization

Many new features and enhancements fall into the category of optimization and performance tuning.

Machine architecture and hardware

Refinements to options `-qarch` and `-qtune`

The compiler option `-qarch` controls the particular instructions that are generated for the specified machine architecture. Option `-qtune` adjusts the instructions, scheduling, and other optimizations to enhance performance on the specified hardware. These options work together to generate application code that gives the best performance for the specified architecture. Skillful use of these options in combination is key to achieving maximal exploitation of IBM processors and hardware. The coordination of these options has been enhanced in this release to add support for the POWER5 and PowerPC 970 hardware platforms and for greater ease of use.

For a particular architecture specified by `-qarch`, compiling with the default `-qtune` suboption generates code that gives the best performance for that architecture. Option `-qarch` can now specify a group of architectures; compiling with `-qtune=auto` generates code that runs on all of the architectures in the specified group, but the instruction sequences will be those with the best performance on the architecture of the compiling machine.

New built-in functions for POWER5 processors

The following built-in functions are available on all PowerPC systems. On POWER5 systems, these functions use POWER5 instructions to take advantage of the POWER5 hardware. All supported built-in functions are described in *XL C/C++ Compiler Reference*.

New built-in functions for all PowerPC systems

Function	Description
<code>int __popcnt4(unsigned int);</code>	Returns the number of bits set (=1) for a 32-bit integer.
<code>int __popcnt8(unsigned long long);</code>	Returns the number of bits set (=1) for a 64-bit integer.
<code>int __poppar4(unsigned int);</code>	Returns 1 if an odd number of bits is set for a 32-bit integer. Otherwise, returns 0.
<code>int __poppar8(unsigned long long);</code>	Returns 1 if an odd number of bits is set for a 64-bit integer. Otherwise, returns 0.
<code>unsigned long __mfspr(const int);</code>	Return a value in the specified special purpose register.
<code>void __mtspr(const int, unsigned long);</code>	Set the special purpose register specified by <code>const int</code> .
<code>unsigned long __mfmsr();</code>	Return the machine state register.
<code>void __mtmsr(unsigned long);</code>	Set the machine state register.

The following built-in functions are available only on POWER5 processors.

Function	Description
<code>double __fre(double);</code>	Returns the result of a floating-point reciprocal operation. The result is a double precision estimate of 1/x.
<code>float __frsqrtes(float);</code>	Returns the result of a reciprocal square root operation. The result is a single precision estimate of the reciprocal of the square root of x.
<code>unsigned long __popcntb (unsigned long);</code>	Counts the 1 bits in each byte of the source operand and places that count into the corresponding byte of the result.
<code>void __protected_unlimited_stream_set_go(unsigned int direction, const void* addr, unsigned int ID);</code>	Establish a protected stream of unlimited length that uses the identifier ID. The stream identifier should be within the range of 0 to 15. The stream begins with the cache line at addr. The stream fetches from either incremental memory addresses or decremental memory addresses, as specified by direction. For incremental memory addresses (that is, a forward direction), the value of direction is 1; for decremental memory addresses, the value of direction is 3. The stream is protected from being replaced by any hardware-detected streams. (Available on PowerPC 970 and POWER5.)
<code>void __protected_stream_set(unsigned int direction, const void* addr, unsigned int ID);</code>	Establish a protected stream of limited length that uses the identifier ID. The stream begins with the cache line at addr and subsequently fetches from either incremental memory addresses or decremental memory addresses, as specified by direction. The stream is protected from being replaced by any hardware-detected streams.
<code>void __protected_stream_count(unsigned int unit_cnt, unsigned int ID);</code>	Set the number of cache lines for the limited-length protected stream identified by ID. The number of cache lines is specified by the parameter unit_cnt and should be within the range of 0 to 1023.
<code>void __protected_stream_go();</code>	Start to prefetch all limited-length protected streams.
<code>void __protected_stream_stop(unsigned int ID);</code>	Stop prefetching the protected steam identified by ID.
<code>void __protected_stream_stop_all();</code>	Stop prefetching all protected steams.

New built-in functions for floating-point division

Four new built-in functions for floating-point division are included in this release. These software implementations of floating-point division algorithms take advantage of the PowerPC architecture and can be significantly faster than corresponding hardware instructions when used in a vector context. The new built-ins are supported for all PowerPC processors, including POWER5.

Hardware division instructions are obtained by default if floating-point division is coded in the source program, but the compiler makes the choice between the hardware or software division code, depending on which it deems faster. The new built-in functions allow the user to explicitly invoke the software algorithms. The default rounding mode (round-to-nearest) must be in effect when the routines are called.

Built-in functions for floating-point division

Function	Description
double __swdiv_nochk(double, double);	Floating-point division of double types; no range checking. Argument restrictions: numerators equal to infinity, or denominators equal to infinity, zero, or denormalized are not allowed.
double __swdiv(double, double);	Floating-point division of double types. No argument restrictions.
float __swdivs_nochk(float, float);	Floating-point division of float types; no range checking. Argument restrictions: numerators equal to infinity, or denominators equal to infinity, zero, or denormalized are not allowed.
float __swdivs(float, float);	Floating-point division of double types. No argument restrictions.

New XL C/C++ pragmas

Pragma directives are described in detail in *XL C/C++ Compiler Reference*.

Pragma	Description
#pragma unrollandfuse	A pragma for optimizing nested for loops. Instructs the compiler to replicate the body of the outer loop, which is itself a loop nest, and to fuse the replicas into a single unrolled loop nest.
#pragma stream_unroll	Breaks a stream contained in a for loop into multiple streams. Intended for loops that have a large iteration count and a small number of streams.
#pragma block_loop	Instructs the compiler to create a blocking loop for a specific for loop in a loop nest. Blocking a loop involves dividing the iteration space of a loop into parts or blocks. An additional outer loop is created, known as the <i>blocking loop</i> , which drives the original loop for each block.
#pragma loopid	Marks a for loop with a scope-unique identifier. The identifier can be used by #pragma block_loop and other pragmas to control the transformations on that loop and to provide information on the loop transformations through the use of option -qreport. The identifier can also be used to identify blocking loops.
#pragma disjoint	C++ implementation added.
extensions to #pragma unroll	Loop unrolling consists of replicating the body of a loop in order to reduce the number of iterations required to complete the loop. The #pragma unroll directive indicates to the compiler that the for loop that immediately follows the directive can be unrolled. The functionality of this pragma has been extended to allow it to be applied to both the innermost and outermost for loops. The extended #pragma functionality still excludes application to for loops that have alternate entry points.

New optimization utilities

This release contains two new utilities related to the profile-directed feedback (PDF) compilation process. Through the use of profile-directed feedback, the compiler can provide an optimized executable that reflects how that executable ran

in a number of different scenarios. A PDF record is produced as a side effect of running the instrumented executable in one of these scenarios. These records constitute the data that are collated to define typical program behavior.

The **showpdf** command provides the ability to display the call and block counts for all procedures executed in a profile-directed feedback training run. The utility requires compilation under the options `-qpdf1` and `-qshowpdf`.

The **mergepdf** command allows the user to specify the relative importance of two or more PDF records and to combine them into a single record. This allows the user to compensate for training runs with higher execution counts (that is, longer run time), which would otherwise dominate the profile data.

IBM Mathematics Accelerated Subsystem (MASS) libraries

Starting with Version 7.0, XL C/C++ ships the IBM Mathematical Accelerated Subsystem (MASS) libraries of tuned mathematical intrinsic functions. The MASS scalar library, `libmass.a`, contains an accelerated set of frequently used math intrinsic functions in the AIX system library `libm.a`. The MASS vector libraries `libmassv.a`, `libmassvp3.a`, and `libmassvp4.a` contain tuned and thread-safe intrinsic functions that can be used with either Fortran or C applications. The general vector library, `libmassv.a`, contains vector functions that will run on all computers in the IBM pSeries and RS/6000 families, while `libmassvp3.a` and `libmassvp4.a` each contain a subset of `libmassv.a` functions that have been specifically tuned for the POWER3 and POWER4 processors, respectively.

SMP thread binding

Shared memory parallelization (SMP) is implemented by creating user threads that are scheduled to run on kernel threads by the operating system. For some workloads, binding threads to processors can improve performance by avoiding the costs of thread migration. Currently, threads created by the SMP run time are not bound to any particular processor, and the AIX operating system takes care of the scheduling of a thread. With the SMP thread binding feature, programs that are dynamically linked to the SMP run time can have their threads bound to processors as specified by the user.

SMP thread binding employs a two-part option, set on the `XLSMPOPT` environment variable. The user specifies the CPU ID of the first thread to be bound and the number of processors to advance (*stride*) from the current processor to bind the next thread.

The ability to specify the starting CPU ID is advantageous when multiple OpenMP programs are running on the same machine. If every OMP program binds its threads starting from CPU 0, an imbalance occurs, in which the first few CPUs in the system are loaded, while the remaining CPUs are not used at all. Allowing the user to specify a stride of 2 allows a non-HPC system to have its threads bound to even-numbered CPU ids, assuming a start at CPU 0. Binding to even-numbered CPUs provides threads to the full L2 cache and bandwidth.

Programs that use processor bindings should become Dynamic Logical Partitioning (DLPAR)-aware. For more information on DLPAR awareness, see *General Programming Concepts: Writing and Debugging Programs*, part of the AIX system documentation.

Conformance to industry standards

This section describes new features implemented by XL C/C++ to conform to various industry standards.

ISO/IEC 14882:2003(E) Programming languages -- C++

XL C/C++ conforms to the revised international C++ standard ISO/IEC 14882:2003(E), Programming languages -- C++.

Starting in Version 7, XL C++ adds support for unordered associative containers, in conformance with *Draft Technical Report on Standard Library Extensions* (TR1). The hash functions and new hash-based containers added as extensions to the C++ Standard Library are as follows:

Header file	Addition
standard header <functional>	functional template <code>std::tr1::hash</code>
new header <unordered_set>	container <code>std::tr1::unordered_set</code> container <code>std::tr1::unordered_multiset</code>
new header <unordered_map>	container <code>std::tr1::unordered_map</code> container <code>std::tr1::unordered_multimap</code>

TR1 libraries are declared in nested namespace `std::tr1`. The new XL C++ TR1 library components are enabled by defining the macro `__IBMCPP_TR1__`.

OpenMP API V2.0 support for C, C++, and Fortran

The OpenMP Application Program Interface (API) is a portable, scalable programming model that provides a standard interface for developing multiplatform, shared-memory parallel applications in C, C++, and Fortran. The specification is defined by the OpenMP organization, a group of major computer hardware and software vendors, which includes IBM. XL C/C++ Enterprise Edition for AIX is compliant with OpenMP Specification 2.0: the compiler recognizes and preserves the semantics of the following OpenMP V2.0 elements:

- Comma delimiter for multiple clauses in the `#pragma omp` directive.
- The `num_threads` clause.
- The `copyprivate` clause.
- `threadprivate` static block scope variables.
- Support for C99 variable length arrays.
- Redundant declaration of private variables.
- Timing routines `omp_get_wtime` and `omp_get_wtick`.

Enhanced Unicode and NLS support

As recommended in a recent report from the C Standard committee, the C compiler extends C99 to add new data types to support UTF-16 and UTF-32 literals. The C++ compiler also supports these new data types for compatibility with C.

Also new in this release, the C++ runtime is able to use the ability of the AIX V5.2 operating system to load multiple locales if the application runs on such a system.

Support for Boost libraries

The XL C++ compiler delivers a high level of compatibility with the 1.30.2 Boost libraries. These libraries were created to provide a set of reusable, Open Source C++ libraries that are suitable for standardization. For more information, see the Boost web site at <http://www.boost.org>.

Language extensions related to GNU C and C++

The GNU C extensions to C99 and the GNU C++ extensions to Standard C++ are not industry standards. Nevertheless, these non-proprietary language features from the Open Source community have attained a certain currency. XL C/C++ implements a subset of the GNU C and C++ extensions. Support for the following GNU C features has been added in this release.

Feature	Remarks
Labels as Values	Including computed goto statements. This feature is now fully compatible with the GNU C implementation.
Type Attributes	Attribute aligned and attribute packed. ▶ C
Function Attributes	Attributes format, format_arg, always_inline, noinline.
Alternate Keywords	Internal changes to implementation of __extension__ .
Nested Functions	▶ C C support only.
Cast to a Union Type	▶ C C support only.
Macros with a Variable Number of Arguments	Using an identifier in place of __VA_ARGS__ and removing trailing comma when no __VA_ARGS__ arguments are specified.
gcc Inline Assembler Instructions with C Expression Operands	Partial support only.
GNU C Complex Types	C++ support added.
GNU C Hexadecimal Float Constants	C++ support added.
C99 Compound Literals	C++ support added.
Arrays of Length Zero	C++ support added.
Variable Length Arrays	C++ support added.

Accommodation of third-party C++ run-time libraries

The C++ compiler can compile C++ applications so that the application supports only the core language, thus enabling it to link with C++ run-time libraries from third-party vendors. The following archive files enable this functionality.

lib*C*core.a	Contains exception handling, RTTI, static initialization, new and delete operators. Does not contain any of the following libraries: Input/Output, Localization, STL Containers, Iterators, Algorithms, Numerics, Strings.
libCcore.a	The core language version of the C++ run-time library, libC.a.

libC128core.a	The core language version of libC128.a.
libhCcore.a	The language core version of libhC.a.

The following invocation commands have been added to facilitate using these archives:

x1Ccore	x1Ccore_r	x1Ccore_r7
x1C128core	x1C128core_r	x1C128core_r7

Ease of use

New C++ compiler invocation

The compiler invocation `xlc++` has been added for portability among all supported platforms. The invocation is equivalent to the invocation `x1C` on all platforms and is recommended. However, `x1C` is still fully supported.

Documentation

XL C/C++ ships with an Information Center of searchable HTML files. The search engine of the new help system is believed to produce hits with greater relevance per search than that of previous releases. The Information Center can be installed on an intranet and accessed by pointing the browser to `http://server_name:5312/help/index.jsp`. The product help system is also viewable online at `http://www.ibm.com/software/awdtools/vacpp/library`.

Starting in Version 7, a man page is provided for the compiler invocation commands and for each command-line utility. The man page for the compiler invocations replaces the text help file provided in earlier releases.

Template registry enhancement

The C++ compiler uses a batch template instantiation scheme that involves a registry of template instantiations. In this release, the compiler adds versioning information to the template registry file that is created. This information is used by the compiler internally to track which version of the template registry file format should be used.

New XL C/C++ options

New and changed compiler options are described in detail in the *XL C/C++ Compiler Reference*.

Option	Description and remarks
<code>-qasm=gcc</code>	Enables partial support for assembler instructions with C expression operands. Instructs the compiler to recognize the <code>asm</code> keyword and its alternate spellings and to use the gcc syntax and semantics for the keyword. The default is <code>-qnoasm</code> .
<code>-qasm_as</code>	Specifies the path and flags used to invoke an alternate assembler program in order to handle the code in an <code>asm</code> directive. This option overrides the default setting of the <code>as</code> command defined in the compiler configuration file.

Option	Description and remarks
-qdirectstorage	Asserts that write-through enabled or cache-inhibited storage may be referenced in a given compilation unit. The intention of this option is to avoid unexpected behavior due to different storage control attributes that are allowed by the PowerPC architecture. The default is -qnodirectstorage .
-qkeepparm	Ensures that the parameters of a function passed in registers are saved onto the stack, instead of possibly being moved to different memory locations to improve performance. The default is -qnokeepparm .
-qnoprefetch	Instructs the compiler not to automatically insert software prefetch instructions, thus allowing the user to turn off this aspect of optimization. The default is -qprefetch .
-qnotrigraph	Instructs the compiler not to interpret trigraph sequences, regardless of the specified language level. On AIX, the default is -qtrigraph .
-qroptr	Instructs the compiler to allow read-only pointers to be moved from the .data section to the .text section. The .text section is always read-only and is never modified by either the loader or by an application. A reduction in memory usage is possible if the constant pointers of an application are allowed to be moved from the .data section into the .text section, which is shared among multiple processes. Code produced with -qroptr in effect is not valid for shared libraries. The default is -qnoroptr , which leaves the read-only pointers in the .data section.
-qsaveopt	Instructs the compiler to save the command-line options against which a source file is compiled into the corresponding object file. The option has no effect if compilation does not result in a .o file. The default is -qnosaveopt .
-qshowpdf	When specified with -qpdf1 , the compiler inserts additional profiling information into the compiled application to collect call and block counts for all procedures in the application. Running the compiled application records the call and block counts to the file <code>._pdf</code> . The contents of <code>._pdf</code> can then be retrieved with the showpdf utility. The default is -qnoshowpdf .
-qsourcecype	Controls the interpretation of input file names. The default behavior is that the programming language of a source file is implied by the suffix of its file name. The default is -qsourcecype .
-qweaksymbol	Instructs the compiler to generate weak symbols for inline functions with external linkage, identifiers specified as weak with <code>#pragma weak</code> , or functions specified as weak with <code>__attribute__((weak))</code> . The option can also be used to suppress linker messages warning of duplicate symbols when compiling C++ programs containing extern inline functions. The default is -qnoweaksymbol .
-qutf	Enables the recognition of UTF literal syntax which provides 16- and 32-based string literals for Unicode encoding forms.
-qflttrap=nanq	Instructs the compiler to generate extra instructions in the code to trap NaNQs (Not a Number Quiet). The intent is to detect all NaNQs handled by or generated by floating point instructions, including those created by valid operations.

Option	Description and remarks
-qipa=infrequentlabel	Specifies a list of labels that are likely to be called infrequently during the course of a typical program run. The compiler can make other parts of the program faster by doing less optimization for calls to these labels. This option is only applicable to user-defined labels.
-qlanglvl=newexcp	The -qlanglvl=newexcp suboption determines the data type of the exception thrown by an allocation function that fails to allocate storage. When the suboption is in effect, an allocation function declared with a non-empty throw expression throws an exception of class <code>std::bad_alloc</code> or a class derived from <code>std::bad_alloc</code> if it fails to allocate storage. This behavior conforms to the C++ standard. When this suboption is not explicitly specified, any allocation function that fails to allocate storage returns a null pointer, regardless of whether it has an exception specification.
-qshowinc suboptions	Used with -qsource to selectively show user header files or system header files in the program source listing. The default is -qnoshowinc . New suboptions allow more specific control over which header files are included. The individual suboptions are <code>all</code> , <code>usr</code> (include user header files), <code>nousr</code> (exclude user header files), <code>sys</code> (include system header files), and <code>nosys</code> (exclude system header files). Multiple suboptions can be specified using a colon delimiter.

Customizing the compilation environment

This section discusses the mechanisms used by XL C/C++ to specify the search paths for directories containing include files and libraries. The mechanisms are environment variables, symbolic links, and configuration files.

Environment variables

Part of the compilation environment are the search paths for special files such as libraries and include files. The following system variables are used by the compiler.

LIBPATH Specifies the directory path for dynamically loaded libraries. Correlates to the LD_LIBRARY_PATH variable on other systems. Affects the link-editor and system loader.

If LIBPATH is set in the environment, the value is read and used by the **ld** command. If a linking operation occurs, either directly or as part of one of the compiler invocations, the value of LIBPATH is used to search for dependent libraries, and the contents of the variable are stored in the resulting module. In the absence of **-L** command-line options, the contents of LIBPATH are prepended to the default library search path.

LIBPATH is ignored at link time if the command line contains any use of the **-L** option. The paths specified on the command line by one or more **-L** options are concatenated in the order in which they appear. The composite path specification is prepended to the default library search and stored in the loader section of the constructed module.

At run time, LIBPATH is used in different ways by the system loader, depending on whether dynamic loading is involved.

The LIBPATH environment variable is checked at exec time and is cleared when the privilege of the program is different from that of the user invoking the program. A process designed to run with an alternate effective user or group ID should find dependent modules only from trusted locations that are embedded within the application.

MANPATH Specifies the directory path for the product man pages.

NLSPATH Specifies the directory path of National Language Support libraries.

PATH Specifies the directory path for the executable files of the compiler.

PDFDIR Specifies the directory in which the profile data file is created. The default value is unset, and the compiler places the profile data file in the current working directory. Setting this variable to an absolute path is recommended for profile-directed feedback.

TMPDIR Specifies the directory in which temporary files are created. The default location may be inadequate at high levels of optimization, where temporary files can require significant amounts of disk space.

Create symbolic links for the path

The command-line interfaces for XL C/C++ are not automatically installed in /usr/bin. To invoke the compiler without having to specify the full path, do one of the following steps:

- Create symbolic links for the specific driver contained in /usr/vacpp/bin and /usr/vac/bin to /usr/bin.
- Add /usr/vac/bin and /usr/vacpp/bin to the PATH environment variable.

Configuration files

A configuration file is a plain text file that specifies options that are read every time you run the compiler. The name of a configuration file ends with a .cfg file name extension.

You can instruct the compiler to use a particular configuration file by invoking the compiler with the **-F** option and specifying the fully qualified file name.

The compiler option **-I** *directory_name* allows you to add directories to the search paths in the configuration file. The configuration file itself uses the **-I** option internally to set the directory paths that it controls. The compiler searches the directories specified by **-I** within the configuration file before searching those specified by **-I** options on the command line.

See *XL C/C++ Compiler Reference* for more information.

Controlling the compilation process

The overall compilation process consists of three phases: preprocessing, translation to object code, and linking. By default, a compiler invocation command invokes all phases of the compilation process to translate a program from source code to executable output. If file names for input and output files are specified when the compiler is invoked, it determines the starting and ending phases from the file name suffix (extension) of the input and output files.

You can also create a particular type of output file at any compilation phase by using appropriate compiler options. For example, invoking the `xlc` or `xlC` command with the `-E` or `-P` option performs only the preprocessing phase on the input files. The compiler invocation determines from the extension of the input file name whether to call the compiler, the assembler, or the linker.

Related References

- `-qphsinfo` compiler option in *XL C/C++ Compiler Reference*

Invoking the compiler

XL C/C++ provides a selection of base compiler invocation commands, which support various version levels of the C and C++ languages. Each invocation command automatically sets a compiler suboption for language level, options for other related language features, and any related predefined macros. In most cases, you should use the `xlc` command to compile C source files and the `xlC` command to compile C++ source files, or when you have both C and C++ source files.

Base invocation commands

<code>xlc</code>	<code>cc</code>	<code>xlc++core</code>
<code>xlc++</code>	<code>c89</code>	<code>xlCcore</code>
<code>xlC</code>	<code>c99</code>	

The invocation `xlc++` is equivalent to `xlC`.

Variations of a base command are provided to support the requirements of special environments and file systems. A variation is formed by attaching a suffix to the base command.

Suffix	Description
<code>_r</code> <code>_r4</code> <code>_r7</code>	Used for compiling thread-safe applications. Also referred to as <i>reentrant compiler invocations</i> .
<code>128</code> <code>128_r</code> <code>128_r4</code> <code>128_r7</code>	Increases the length of long double types from 64 to 128 bits and links with the 128-bit versions of the C and C++ runtimes. When attached to the base invocations <code>c89</code> or <code>c99</code> , the suffix is preceded by an underscore (for example, <code>c99_128</code>).

In addition, the `gxlc` and `gxlc++` utilities are specialized compiler invocations.

Object model

An object model describes how C++ object and helper data structures are laid out in memory. It may also affect name mangling. Two object models are supported on the AIX platform: `compat` and `ibm`. These differ in the way the compiler lays out the virtual function table, the type of support provided for virtual base classes, and the name mangling scheme used. The `compat` object model creates a run-time module that will be compatible with other compiled with the same model or with previous versions of the compiler. The `ibm` object model can provide improved performance, especially if the source contains class hierarchies with many virtual base classes. This object model tends to create a smaller derived class and faster access to the virtual function table.

Types of input and output files

The compiler uses the file name extension to determine the appropriate compilation phase and invoke the associated tool.

The compiler accepts the following types of files as input:

Accepted input file types

File type description	File name extension	Example
C and C++ source file	.c (lowercase <i>c</i>) for C language source files; .C (uppercase <i>c</i>), .cc, .cp, .cpp, .cxx, .c++ for C++ source files	<i>file_name.c</i> <i>file_name.C</i> , <i>file_name.cc</i> , <i>file_name.cpp</i> , <i>file_name.cxx</i> , <i>file_name.c++</i>
Preprocessed source file	.i	<i>file_name.i</i>
Object file	.o	hello.o
Assembler file	.s	check.s
Archive file	.a	v1r5.a
Loadable module or shared library file	.so	my_shrllib.so
IPA control files (<code>-qipa=file_name</code>)	No naming convention for <i>file_name</i> is enforced.	ipactl

You can specify the following types of output files when invoking the compiler:

Types of output files

File type description	Example
Executable file	By default, a.out
Object files	<i>file_name.o</i>
Loadable module or shared object file	<i>file_name.so</i>
Assembler files	<i>file_name.s</i>
Preprocessed files	<i>file_name.i</i>
Listing files	<i>file_name.lst</i>
Target file	<i>file_name.u</i>

Related References

- `-qsource` compiler option in *XL C/C++ Compiler Reference*

Default behavior

If you invoke the compiler without specifying any options, the behavior of the compiler is governed by the following default settings:

- Attempts to read and invoke the options specified in a configuration file.
- Aligns structures using the default alignment, `-qalign=power`.
- Produces an unoptimized executable named `a.out` in the current directory.

See *XL C/C++ Compiler Reference* for more information.

Getting started with compiler options

Compiler options perform a variety of functions, such as setting compiler characteristics, describing the object code to be produced, controlling the diagnostic messages emitted, and performing some preprocessor functions. You can specify compiler options on the command line, in a configuration file, in your source code, or any combination of these techniques. Most options that are not explicitly set take the default settings.

When multiple compiler options have been specified, it is possible for option conflicts and incompatibilities to occur. To resolve these conflicts in a consistent fashion, the compiler applies the following priority sequence unless otherwise specified:

1. Source file *overrides*
2. Command line *overrides*
3. Configuration file *overrides*
4. Default settings

Generally, among multiple command-line options, the last specified prevails.

Note: The **-I** compiler option is a special case. The compiler searches any directories specified with **-I** in the `vac.cfg` file *before* it searches the directories specified with **-I** on the command line. The option is cumulative rather than preemptive.


Related References

- See *XL C/C++ Compiler Reference* for more information.

Compiler messages

XL C/C++ uses a five-level classification scheme for diagnostic messages. Each level of severity is associated with a compiler response. Not every error halts compilation. The following table provides a key to the abbreviations for the severity levels and the associated compiler response.

Severity levels and compiler response

Letter	Severity	Compiler Response
I	Informational	Compilation continues. The message reports conditions found during compilation.
W	Warning	Compilation continues. The message reports valid, possibly unintended conditions.
 E	Error	Compilation continues and object code is generated. Error conditions exist that the compiler can correct, but the program might not produce the expected results.
S	Severe error	Compilation continues, but object code is not generated. Error conditions exist that the compiler cannot correct.

Severity levels and compiler response

Letter	Severity	Compiler Response
U	Unrecoverable error	The compiler halts. An unrecoverable error has been encountered. If the message indicates a resource limit (for example, file system full or paging space full), provide additional resources and recompile. If it indicates that different compiler options are needed, recompile using them. If the message indicates an internal compiler error, the message should be reported to your IBM service representative.

The default behavior of the compiler is to compile with the option **-qnoinfo** or **-qinfo=noall**. The suboptions for **-qinfo** provide the ability to specify a particular category of informational diagnostics. For example, **-qinfo=por** limits the output to those messages related to portability issues.

Note: In C, the option **-qinfo** specified without suboption is equivalent to **-qinfo=all**; in C++, **-qinfo** specified without suboption is equivalent to **-qinfo=all:noppt**.

Return codes

At the end of compilation, the compiler sets the return code to zero under any of the following conditions:

- No messages are issued.
- The highest severity level of all errors diagnosed is less than the setting of the **-qhalt** compiler option, and the number of errors did not reach the limit set by the **-qmaxerr** compiler option.
- No message specified by the **-qhaltmsg** compiler option is issued.

Otherwise, the compiler sets one of the return codes documented in *XL C/C++ Compiler Reference*.

Compiler message format

By default, diagnostic messages have the following format:

"file", line line_number.column_number: 15cc-nnn (severity) message_text.

where 15 is the compiler product identifier, *cc* is a two-digit code indicating the compiler component that issued the message, *nnn* is the message number, and *severity* is the letter of the severity level. The possible values for *cc* are:

- 00 Code generating or optimizing message
- 01 Compiler services message
- 05 Message specific to the C compiler
- 06 Message specific to the C compiler
- 40 Message specific to the C++ compiler
- 86 Message specific to interprocedural analysis (IPA)

This format is the same as compiling with the **-qnosrcmsg** option enabled. To get an alternate message format in which the source line displays with the diagnostic message, try compiling with **-qsrcmsg** option. Enabling this option instructs the compiler to print to standard error the source line where the compiler thinks the error lies; a second line below it, whenever possible, that points to a specific point in that source line; and the diagnostic message.

Note: Messages are not intended to be used as input to other programs. The message format and content are not intended to be a programming interface and may change from release to release.

Reusing GNU C and C++ compiler options with `gxc` and `gxc++`

Each of the `gxc` and `gxc++` utilities accepts GNU C or C++ compiler options and translates them into comparable XL C/C++ options. Both utilities use the XL C/C++ options to create an `xlc` or `xlc++` invocation command, which they then use to invoke XL C/C++. These utilities are provided to facilitate the reuse of make files created for applications previously developed with GNU C and C++. However, to fully exploit the capabilities of XL C/C++, it is recommended that you use the XL C/C++ invocation commands and their associated options.

The actions of `gxc` and `gxc++` are controlled by the configuration file `gxc.cfg`. The GNU C and C++ options that have an XL C or XL C++ counterpart are shown in this file. Not every GNU option has a corresponding XL C/C++ option. `gxc` and `gxc++` return warnings for input options that were not translated.

The `gxc` and `gxc++` option mappings are modifiable. For information on adding to or editing the `gxc` and `gxc++` configuration file, see “Configuring the option mapping” on page 29.

Example

To use the `gcc -ansi` option to compile the C version of the Hello World program, you can use:

```
gxc -ansi hello.c
```

which translates into:

```
xlc -F:c89 hello.c
```

This command is then used to invoke the XL C compiler.

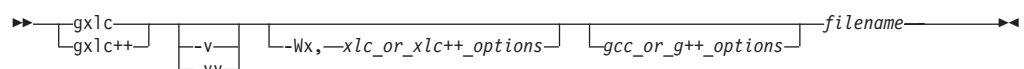
`gxc` and `gxc++` return codes

Like other invocation commands, `gxc` and `gxc++` return output, such as listings, diagnostic messages related to the compilation, warnings related to unsuccessful translation of GNU options, and return codes. If `gxc` or `gxc++` cannot successfully call the compiler, it sets the return code to one of the following values:

- 40** A `gcc` or `g++` option error or unrecoverable error has been detected.
- 255** An error has been detected while the process was running.

`gxc` and `gxc++` syntax

The following diagram shows the `gxc` and `gxc++` syntax:



where:

filename

Is the name of the file to be compiled.

-v Allows you to verify the command that will be used to invoke XL C/C++.

gxc or gxc++ displays the XL C/C++ invocation command that it has created, before using it to invoke the compiler.

-vv Allows you to run a simulation. gxc or gxc++ displays the XL C/C++ invocation command that it has created, but does not invoke the compiler.

-Wx,xlc_or_xlc++_options

Sends the given XL C/C++ options directly to the xlc or xlc++ invocation command. gxc or gxc++ adds the given options to the XL C/C++ invocation it is creating, without attempting to translate them. Use this option with known XL C/C++ options to improve the performance of the utility. Multiple *xlc_or_xlc++_options* use a comma delimiter.







gcc_or_g++_options

Are the gcc or g++ options that are to be translated to xlc or xlc++ options. The utility emits a warning for any option it cannot translate. The gcc and g++ options that are currently recognized by gxc and gxc++ are listed in the configuration file *gxc.cfg*. Multiple *gcc_or_g++_options* are delimited by the space character.



GNU C and C++ to XL C/C++ option mapping

The following table lists the GNU C and C++ options that are accepted and translated by gxc and gxc++. All other GNU options that are specified as input to one of these utilities are ignored or generate an error. If the negative form of a GNU option exists, then the negative form is also recognized and translated by gxc and gxc++.

Mapped options: GNU C and C++ to XL C/C++

GNU C and C++ option	XL C/C++ option
-###	-#
-ansi	-F:c89
-B	-B
-C	-C
-c	-c
-Dmacro[=defn]	-Dmacro[=defn]
-E	-E
-e	-e
-fdollars-in-identifiers	-qdollar
 -fdump-class-hierarchy	-qdump_class_hierarchy
 -fexceptions	-qeh
 -ffor-scope	-qlanglvl=ansifor
 -fno-for-scope	-qlanglvl=noansifor
-ffunction-sections	-qfuncsect
-finline	-qinline
-finline-functions	-qinline
 -fkeep-inline-functions	-qkeepinlines
 -fno-gnu-keywords	-qnokeyword=typeof










Mapped options: GNU C and C++ to XL C/C++

GNU C and C++ option	XL C/C++ option
 -fno-operator-names	-qnokeyword=and -qnokeyword=bitand -qnokeyword=bitor -qnokeyword=compl -qnokeyword=not -qnokeyword=or -qnokeyword=xor
-fpascal-strings	-qmacpstr
-fPIC	-qpik=large
-fpic	-qpik=small
 -frtti	-qrtti
-fshort-enums	-qenum=small
-fsigned-bitfields	-qbitfields=signed
-fsigned-char	-qchars=signed
-fstrict-aliasing	-qalias=ansi
-fsyntax-only	-qsyntaxonly
-funroll-all-loops	-qunroll=yes
-funroll-loops	-qunroll=yes
-funsigned-bitfields	-qbitfields=unsigned
-funsigned-char	-qchars=unsigned
-fwritable-strings	-qnoro
-g	-g
-g3	-g
-ggdb	-g
-gxcoff	-g
-I <i>dir</i>	-I <i>dir</i>
-L <i>dir</i>	-L <i>dir</i>
-l <i>library</i>	-l <i>library</i>
-M	-M
-MD	-M
-maix32	-q32
-maix64	-q64
-mcpu=403	-qarch=403
-mcpu=601	-qarch=601
-mcpu=602	-qarch=602
-mcpu=603	-qarch=603
-mcpu=604	-qarch=604
-mcpu=common	-qarch=com
-mcpu=power	-qarch=pwr
-mcpu=power2	-qarch=pwr2
-mcpu=powerpc	-qarch=ppc
-mcpu=powerpc64	-qarch=ppc64
-mcpu=rs64a	-qarch=rs64a

Mapped options: GNU C and C++ to XL C/C++

GNU C and C++ option	XL C/C++ option
-mno-fused-madd	-qfloat=nomaf
-mfused-madd	-qfloat=maf
-mlong-double-64	-qnolonglong
-mlong-double-128	-qlonglong
-mpower	-qarch=pwr
-mpower2	-qarch=pwr2
-mpowerpc	-qarch=ppc
-mpowerpc-gfxopt	-qarch=pwrgr
-mpowerpc64	-qarch=ppc64
-mtune=403	-qtune=403
-mtune=601	-qtune=601
-mtune=602	-qtune=602
-mtune=603	-qtune=603
-mtune=604	-qtune=604
-mtune=common	-qtune=com
-mtune=power	-qtune=pwr
-mtune=power2	-qtune=pwr2
-mtune=powerpc	-qtune=ppc
-mtune=powerpc64	-qtune=ppc64
-mtune=rs64a	-qtune=rs64a
-nodefaultlibs	-qnolib
-nostartfiles	-qnocrt
-nostdinc	-qnostdinc
-nostdlib	-qnolib -qnocrt
-O	-O
-O0	-qnoopt
-O1	-O
-O2	-O2
-O3	-O3
-Os	-O2 -qcompact
-o	-o
-p	-p
-pg	-pg
-r	-r
-S	-S
-s	-s
 -std=c89	-F:c89
 -std=iso9899:1990	-F:c89
 -std=iso9899:199409	-F:c89

Mapped options: GNU C and C++ to XL C/C++

GNU C and C++ option	XL C/C++ option
 -std=c99	-F:c99
 -std=c9x	-F:c99
 -std=iso9899:1999	-F:c99
 -std=iso9899:199x	-F:c99
 -std=gnu89	-qlanglvl=extc89
 -std=gnu99	-qlanglvl=extc99
 -std=gnu9x	-qlanglvl=extc99
 -std=c++98	-qlanglvl=strict98
 -std=gnu++98	-qlanglvl=extended
-time	-qphsinfo
-trigraphs	-qtrigraph
-U <i>macro</i>	-U <i>macro</i>
-u	-u
-Wformat	-qformat
-Wuninitialized	-qinfo=ini
-Wunreachable-code	-qinfo=eff
-W <i>a,option</i>	-W <i>a,option</i>
-W <i>l,option</i>	-W <i>l,option</i>
-W <i>p,option</i>	-W <i>p,option</i>
-w	-w
-x assembler	-qsourcecetype=assembler
-x c	-qsourcecetype=c
-x c++	-qsourcecetype=c++
-x none	-qsourcecetype=default
-Z	-Z

All other GNU options are ignored and issue an informational message.

Configuring the option mapping

The `gxc` and `gxc++` utilities use the configuration file `gxlc.cfg` to translate GNU C and C++ options to XL C/C++ options. Each entry in `gxlc.cfg` describes how the utility should map a GNU C or C++ option to an XL C/C++ option and how to process it.

An entry consists of a string of flags for the processing instructions, a string for the GNU C option, and a string for the XL C/C++ option. The three fields must be separated by whitespace. If an entry contains only the first two fields and the XL C/C++ option string is omitted, the GNU C option in the second field will be recognized by `gxc` and silently ignored.

The # character is used to insert comments in the configuration file. A comment can be placed on its own line, or at the end of an entry.

The following syntax is used for an entry in `gxl.c.cfg`:

```
abcd    "gcc_or_g++_option"    "xl.c_or_xl.c++_option"
```

where:

a Lets you disable the option by adding `no-` as a prefix. The value is either *y* for yes, or *n* for no. For example, if the flag is set to *y*, then `finline` can be disabled as `fno-inline`, and the entry is:

```
ynn*    "-finline"            "-qinline"
```

If given `-fno-inline`, then `gxl.c` will translate it to `-qnoinline`.

b Informs the utility that the XL C/C++ option has an associated value. The value is either *y* for yes, or *n* for no. For example, if option `-fmyvalue=n` maps to `-qmyvalue=n`, then the flag is set to *y*, and the entry is:

```
ynn*    "-fmyvalue"          "-qmyvalue"
```

`gxl.c` and `gxl.c++` will then expect a value for these options.

c Controls the processing of the options. The value can be:

- *n*, which tells the utility to process the option listed in the `gcc-option` field
- *i*, which tells the utility to ignore the option listed in the `gcc-option` field. `gxl.c` and `gxl.c++` will generate a message that this has been done, and continue processing the given options.
- *e*, which tells the utility to halt processing if the option listed in the `gcc-option` field is encountered. `gxl.c` and `gxl.c++` will also generate an error message.

For example, the `gcc` option `-I-` is not supported and must be ignored by `gxl.c` and `gxl.c++`. In this case, the flag is set to *i*, and the entry is:

```
nni*    "-I-"
```

If `gxl.c` and `gxl.c++` encounters this option as input, it will not process it and will generate a warning.

d Lets `gxl.c` and `gxl.c++` include or ignore an option based on the type of compiler. The value can be:

- *c*, which tells `gxl.c` and `gxl.c++` to translate the option only for C.
- *x*, which tells `gxl.c` and `gxl.c++` to translate the option only for C++.
- ***, which tells `gxl.c` and `gxl.c++` to translate the option for C and C++.

For example, `-fwritable-strings` is supported by both compilers, and maps to `-qno`. The entry is:

```
nnn*    "-fwritable-strings"    "-qno"
```

"gcc_or_g++_option"

Is a string representing a `gcc` or `g++` option supported by GNU C, Version 3.3. This field is required and must appear in double quotation marks.

"xl.c_or_xl.c++_option"

Is a string representing an XL C/C++ option. This field is optional, and, if present, must appear in double quotation marks. If left blank, `gxl.c` and `gxl.c++` ignores the *gcc_or_g++_option* in that entry.

It is possible to create an entry that will map a range of options. This is accomplished by using the asterisk (*) as a wildcard. For example, the gcc -D option requires a user-defined name and can take an optional value. It is possible to have the following series of options:

```
-DCOUNT1=100  
-DCOUNT2=200  
-DCOUNT3=300  
-DCOUNT4=400
```

Instead of creating an entry for each version of this option, the single entry is:

```
nni*      "-D*"      "-D*"
```

where the asterisk will be replaced by any string following the -D option.

Conversely, you can use the asterisk to exclude a range of options. For example, if you want gxc or gxc++ to ignore all the -std options, then the entry would be:

```
nni*      "-std*"
```

When the asterisk is used in an option definition, option flags *a* and *b* are not applicable to these entries.

The character % is used with a GNU C or GNU C++ option to signify that the option has associated parameters. This is used to insure that gxc or gxc++ will ignore the parameters associated with an option that is ignored. For example, the -include option is not supported and uses a parameter. Both must be ignored by the application. In this case, the entry is:

```
nni*      "-include %"
```

Related References

- The GNU Compiler Collection online documentation at <http://gcc.gnu.org/onlinedocs/>

Options summary: C compiler

This chapter appendix presents a summary of the C compiler options, grouped by type. The higher level groupings contain subgroups of options. In addition to a subgroup for basic translation of source code, one subgroup comprises options for special handling or control of the code, such as adding specialized debugging information. Another subgroup pertains to control of the linker and library search paths. Options related to performance and optimization are summarized at the end of chapter “Getting started with optimization” on page 35. For description, full option syntax, and usage of each option, see *XL C/C++ Compiler Reference*.

Basic translation

The options in this grouping have the broadest applicability for basic translation of source code. The subgroups of compiler options are generally concerned with:

- Standards compliance.
- Compilation mode or control of the compiler driver.
- Manipulating the source code for code generation.
- Generating specialized diagnostics.
- Manipulating the compiled code.

Options related to basic translation of source code

Standards compliance	Compilation mode or control of compiler driver
-qgenproto, -qno-genproto -qlanglvl -qlibansi, -qno-libansi	-# -q32 -q64 -F -qpath -qproto, -qno-protocol -qsource-type
Source code generation	
-qalloca -qasm, -qnoasm -qasm_as -qattr, -qnoattr -B -C -qcpluscmt, -qno-cplusplus -D -qdbc, -qno-dbc -qdigraph, -qno-digraph -qdirectstorage, -qno-directstorage -E -qfuncsect, -qno-funcsect -qignprag -M -ma	-qmacpstr, -qno-macpstr -qmakedep -qmbcs, -qno-mbcs -P -qpascal, -qno-pascal -qroptrs, -qno-roptrs -qsmallstack, -qno-smallstack -qsyntaxonly -t -qtabsize -qtrigraph, -qno-trigraph -U -qutf, -qno-utf -W -qweaksymbol, -qno-weaksymbol
Diagnostics	Compiled code
-qflag -qinfo, -qno-info -qmaxerr, -qno-maxerr -qphsinfo, -qno-phsinfo -qprint, -qno-print -qshowinc, -qno-showinc -qsource, -qno-source -qsrcmsg, -qno-srcmsg -qsuppress, -qno-suppress -V -v -w -qwarn64, -qno-warn64 -qxcall, -qno-xcall	-qbitfields -c -qchars -qdataimported -qdatalocal -qdollar, -qno-dollar -qexpfile -o -qprocimported -qprocllocal -qprocunknown -S -qstatsym, -qno-statsym -qtbtable -qpconv, -qno-pconv

Special handling and control

The options in this grouping provide fine-grain control of the translation process and have less general applicability than basic translation options. The topics within this grouping of compiler options are generally concerned with:

- Data alignment.
- Compilation mode or control of the compiler driver.
- Manipulating the source code for code generation.
- Generating specialized diagnostics.
- Manipulating the compiled code.

Options for special handling, fine tuning, and debugging

Data alignment	Parallelization
-qalign -qenum	-qsmp, -qnosmp -qthreaded, -qnothreaded
Floating-point and numerical features	
<i>Sizes</i> -qldb1128, -qnoldb1128 -qlongdouble, -qno longdouble -qlonglit, -qno longlit -qlonglong, -qno longlong	<i>Rounding of floating-point values</i> -qrndflt, -qnorndflt -y
<i>Single-precision values</i> -qhsflt, -qnohsflt -qhsngl, -qnohsngl	<i>Other floating-point options</i> -qfloat -qfltrap, -qnofltrap -qmaf, -qnomaf
Debugging	
-qcheck, -qnocheck -qdpcl, -qnodpcl -qdbxextra, -qnodbxextra -qextchk, -qnoextchk -qfullpath, -qnofullpath -g -qhalt -qheapdebug, -qnoheapdebug -qinitauto, -qnoinitauto	-qkeeparm, -qnokeeparm -qlinedebug, -qno linedebug -qlist, -qno list -qlistopt, -qno listopt -qsaveopt, -qnosaveopt -qsymtab -qxref, -qnoxref

Linking and library-related options

The options in this grouping are related to the linking phase of the compilation process. This grouping also contains options that provide specialized ways to specify search paths for finding libraries and header files. These compiler options are generally concerned with:

- Placing string literals and constants.
- Static and dynamic linking and libraries.
- Specifying search directories.

Options for controlling the ld command

Placing string literals and constants	Static and dynamic linking and libraries
-qkeyword, -qnokeyword -qro, -qnor -qroconst, -qnorconst	-b -brtl -e -G -qmkshrobj -qstdinc, -qnostdinc
Search directories	Other linker options
-I -L -l (lowercase el) -qidirfirst, -qno idirfirst -r -Z	-f -qinlglue, -qnoinglue

Options summary: C++ compiler

Most of the C compiler options are available for compiling C++ programs. The following table presents additional compiler options specific to compiling C++ programs and the C options that are *not* available for compiling C++ programs on the AIX platform:

Compiler options for C++ programs

C++-specific options	C-only options
-+	-qalloca
-qcinc	-qassert, -qnoassert
-qeh, -qnoeh	-qc_stdinc
-qhaltormsg	-qcplusplus, -qnocplusplus
-qkeepinlines, -qnokeepinlines	-qdbxextra, -qnodbxextra
-qnamemangling	-qgenproto, -qnoegenproto
-qobjmodel	-ma
-qoldpassbyvalue, -qnooldpassbyvalue	-macpstr, -nomacpstr
-qpriority	-qproto, -qnoproto
-qrtti, -qnortti	-qsrcmsg, -qnosrcmsg
-qstaticinline, -qnostaticinline	-qsyntaxonly
-qtempinc, -qnotempinc	-qupconv, -qnoupconv
-qtemplatercompile, -qnotemplatercompile	
-qtemplaterregistry, -qnotemplaterregistry	
-qtempmax	
-qtplparse	
-qtwolink, -qnotwolink	
-qunique, -qnounique	
-qvftable, -qnovftable	

Getting started with optimization

Simple compilation is a translation or transformation of the source code into an executable or shared object. An optimizing transformation is one that gives your application better overall performance at run time. XL C/C++ provides a portfolio of optimizing transformations tailored to the PowerPC architecture. These transformations can:

- Reduce the number of instructions executed for critical operations.
- Restructure the generated object code to make optimal use of the PowerPC architecture.
- Improve the usage of the memory subsystem.
- Exploit the ability of the architecture to handle large amounts of shared memory parallelization.

Their aim is to make your application run faster.

Significant performance improvements can be achieved with relatively little development effort if you understand the available controls that affect the transformation of well-written code. Programming models such as OpenMP allow you to write high-performance code. This section describes some of the optimizations the compiler can perform to help you balance the trade-offs among run-time performance, hand-coded micro-optimizations, general readability, and overall portability of your source code.

Optimizations are often attempted in the later phases of application development cycles, such as product release builds. If possible, you should test and debug your code without optimization before attempting to optimize it. Embarking on optimization should mean that you have chosen the most efficient algorithms for your program and that you have implemented them correctly. To a large extent, compliance with language standards is directly related to the degree to which your code can be successfully optimized. Optimizers are the ultimate conformance test!

Optimization is controlled by compiler options, directives, and pragmas. However, compiler-friendly programming idioms can be as useful to performance as any of the options or directives. It is no longer necessary nor is it recommended to excessively hand-optimize your code (for example, manually unrolling loops). Unusual constructs can confuse the compiler (and other programmers), and make your application difficult to optimize for new machines.

It should be noted that not all optimizations are beneficial for all applications. A trade-off usually has to be made between an increase in compile time, accompanied by reduced debugging capability, and the degree of optimization done by the compiler.

Related References

- "Using optimization levels" in *XL C/C++ Programming Guide*
- "Optimizing your applications" in *XL C/C++ Programming Guide*

Selected compiler options for optimization

The following table features a selection of basic compiler options for optimizing program performance. For an exhaustive list, see *XL C/C++ Programming Guide*. For documentation of the available suboptions, see *XL C/C++ Compiler Reference* or the options man page.

Table 1. Basic compiler options for optimization

Option	Description
-qnoopt	The compiler performs very limited optimization. This is the default. Before you start optimizing your application, ensure that it compiles successfully with -qnoopt .
-O2	The compiler performs comprehensive low-level optimization, which includes graph coloring, common subexpression elimination, dead code elimination, algebraic simplification, constant propagation, instruction scheduling for the target machine, loop unrolling, and software pipelining.
-qarch -qtune -qcache	The compiler takes advantage of the characteristics of the specific hardware and instruction set where the application run. Use -qarch to specify family of processor architectures for which application code should be generated. Use -qtune to bias optimization toward execution on a given microprocessor. Use -qcache to specify a specific cache or memory geometry.
-qpdf1 -qpdf2	The compiler uses profile-directed feedback to optimize the application based on an analysis of how often different sections of code are typically executed. The PDF process is most useful for applications that contain unstructured branching.
-O3	The compiler performs more aggressive optimization than at -O2 : deeper loop unrolling, better loop scheduling, elimination of the limits on implicit memory usage.
-qhot	The compiler performs high-order transformations, which provide additional loop optimization and optionally performs array padding. This option is most useful for scientific applications that perform a large amount of numerical processing.
-qipa	The compiler performs interprocedural analysis to optimize the entire application as a unit (whole-program analysis). This option is most useful for business applications that contain a large number of frequently used routines. It is also useful for C++ programs with a high level of abstraction. In many cases, this option significantly increases compilation time.
-O4	This is equivalent to -O3 -qipa -qhot -qarch=auto -qtune=auto -qcache=auto . If the compilation takes too long, try compiling with -O4 -qnoipa .
-O5	This is equivalent to -O4 -qipa=level=2 . On the AIX platform, this option also turns on -qhot=vector -qhot=simd , provided that the processor is PowerPC 970 and that AltiVec data types are supported by the operating system.

Getting started with optimization pragmas

A pragma directive is an implementation-specific preprocessor directive that provides the ability to section off a group of lines in the source code of a program to inform the compiler of something about that section. A pragma directive often provides finer-grained control than the similarly named, corresponding compiler

option. For pragma directives related to optimization, the intention is to explicitly notify the compiler of potential opportunities for optimization that it may not be able to detect on its own, or of assumptions it can make, which might facilitate its decisions about optimizing the source code.

XL C/C++ provides pragma directives for performance optimization that have counterparts among XL Fortran directives. Like the Fortran directives, the optimization pragmas can be thought of in descriptive categories: imperative, assertive, and prescriptive. The distinctions among categories are the degree to which the compiler must comply with the instructions or information provided by the pragma.

All OpenMP directives are imperative pragmas. The compiler is required to behave in strict conformance with its implementation of the OpenMP standard. However, the resulting program behavior is not guaranteed to produce correct results. XL C/C++ implements all OpenMP V2.0 pragma directives.

An assertive pragma informs the compiler of an assumption that it can safely make when transforming the lines of source code in which the pragma is in effect. The intention of an assertive pragma is informative, an assertion that imposes no specific obligation on the compiler to behave in a certain way. XL C/C++ provides the following assertive pragmas related to optimization.

Selected assertive #pragma directives

Name	Description
<code>isolated_call(function_list)</code>	Asserts that calls to the named functions do not have side effects.
<code>disjoint(variable_list)</code>	Asserts that none of the named variables have overlapping areas of storage.
<code>ibm independent_loop</code>	Asserts that the following loop has no loop-carried dependencies. This freedom enables locality and parallel transformations.
<code>ibm independent_calls</code>	Asserts that the calls in the following loop do not cause loop-carried dependencies.
<code>ibm permutation</code>	Asserts that elements of the named arrays take on distinct values on each iteration of the following loop. This assertion may be useful in sparse codes.
<code>ibm iterations(iteration_count)</code>	Specifies the number of iterations for the following loop, which helps the compiler to determine if it is advantageous to parallelize the loop. The pragma is applicable to a single loop instead of the entire compilation unit.
<code>execution_frequency(very_low)</code>	Asserts that the control path containing the pragma will be infrequently executed
<code>leaves(function_list)</code>	Asserts that calls to the named functions will not return.

A prescriptive pragma is a suggestion to the compiler to transform the source code under its effects in a particular way in hope of getting better performance. A prescriptive pragma functions like the **inline** keyword: the compiler is not required to implement the suggestion, but it might if doing so is deemed advantageous. XL C/C++ provides the following prescriptive pragmas related to optimization.

Selected prescriptive #pragma directives

Name	Description
ibm sequential_loop	Directs the compiler to execute the following loop in a single thread, even if -qsmp=auto is specified.
ibm snapshot(variable_name)	Sets a debugging breakpoint at the point of the pragma to examine the specified variable.
stream_unroll	Breaks a stream contained in a for loop into multiple streams. Intended for loops that have a large iteration count and a small number of streams.
unroll	Indicates to the compiler that the for loop that immediately follows the directive can be unrolled. The pragma can be applied to both the innermost and outer for loops.
unrollandfuse	Instructs the compiler to replicate the body of the outer loop of a for loop that is itself a loop nest, and fuses the replicas into a single unrolled loop nest.

Related References

- All XL C/C++ pragmas are documented in *XL C/C++ Compiler Reference*.

Porting considerations

This section describes general areas of investigation that can facilitate porting a UNIX-based application to the AIX platform.

Porting an application to run on another platform involves a *source* platform and a *target* platform. At the most basic level, the following question should be considered before doing any coding: *What is being changed as part of the port to the target platform?* A true port involves changing only the hardware and operating system.

In theory, well-written programs that do not rely on platform-specific dependencies, adhere to industry standards, such as POSIX, and conform to standard language definitions without employing nonstandard language extensions, can easily be ported to a new operating system with a minimum of extra work besides recompiling and debugging. When the source platform is a reasonably recent UNIX-based operating system, the changes may be confined to becoming more compliant with industry standards or with a newer version of the same standard. If the application is already running on a Linux system, you have the option to recompile it and run it natively on AIX. Many applications recompile and run without change.

Moreover, compiling an application with different standards-conforming compilers can drive out subtle weaknesses in the source code due to differences in the implementations of the language standards. The result is that the application becomes more robust.

Problems that arise in a porting exercise can be classified into internal and external portability issues. An *internal* portability issue deals with implicit assumptions that are intrinsic to the programming language. For example, C programs assume a particular byte order within integers, a set of relative sizes of integers, and a particular layout of fields within structures. Internal portability pertains to the relationship of the program code to the hardware. This class of porting problems is under a programmer's control.

On the other hand, *external* portability issues pertain to the choice of external interfaces that a program uses, the semantics of these interfaces that are assumed by the program, and the arguments and return values that are passed to and from the program. They deal with libraries and system calls that the program depends upon, code that is invoked by, but external to, the program. External portability can be under a programmer's control if the program uses standardized external interfaces.

Portability issues intrinsic to the language

This section presents some internal portability considerations related to porting to the AIX platform.

- Checking the amount of reliance on GNU C and other language extensions. An application that conforms strictly to its ISO language specification will be maximally portable. IBM XL C/C++ supports a subset of the GNU C and C++ extensions to C and C++. You may need to revisit code that relies on unsupported extensions.

- Checking how null pointers are dereferenced. Some errors in the code can go undetected on a platform due to hardware-dependent characteristics. These kinds of errors might show up when the program is ported to another platform. If AIX is the source platform, you can use the option `-qcheck=nullptr` to help detect such conditions before porting.

The lowest 4K of memory (that is, addresses 0 through 4K-1) are readable and contain zeroes on AIX, but are not readable on the Linux and Mac OS X platforms, and will cause a segmentation violation if accessed on those platforms. For example,

```
if (strcmp(a, NULL) == 0) ...
```

results in a segmentation violation on Linux and Mac OS X, but not on AIX.

- Checking the alignment. The types of alignment supported on AIX are not the same as those supported on the Linux or Mac OS X platforms, even though the names might be the same. If you are porting a program that relies on specific values for `-qalign` or `#pragma align`, you may need to change the program.
- Ensuring the portability of data structures. If you generate data with an application on one platform and read the data with an application on another platform, the data may have an alignment that is different from that which the reading application expects. To avoid this problem, make sure that you use a platform-neutral mechanism for the layout of data in structures. For example, if you enclose a structure with `#pragma pack(1)` and `#pragma pack(pop)` pair, the alignment will be the same on all platforms.
- Using the `gxlc` or `gxlcpp` utility for translating the commands in your makefiles. Not all gcc or g++ options have an XL C/C++ equivalent.
- Using a different compiler invocation mode for 32- or 64-bit applications, or for 128-bit support.
- On Linux or Mac OS X, if the default global operator `new` is called and the allocation request cannot be fulfilled, an exception of type `std::bad_alloc` is thrown. On AIX, the default behavior of the global operator `new` is to return a null pointer if allocation fails.
- Ensuring the portability of applications that use templates. The C++ compiler provides two different methods of working with template files, as alternatives to maintaining templates manually in the source code. Each method has an associated compiler option. The `-qtemplateregistry` compiler option maintains a record of all templates. This method is recommended. An older compiler option, `-qtempinc`, is also provided for applications that you port from another platform. However, on the Mac OS X platform, the compiler option `-qtempinc` is considered deprecated.

Related References

The following IBM Redbooks contain information related to porting. Other Redbooks are available online at www.redbooks.ibm.com.

- *AIX 5L Porting Guide* (2001).
- *Developing and Porting C and C++ Applications on AIX* (2003).

Diagnostics for compile-time errors

A basic recommendation for a porting project is to compile the application with the option **-qinfo=por**. This suboption of **-qinfo** adds diagnostic messages that pertain specifically to portability. The option **-qinfo=warn64** instructs the compiler to emit diagnostic messages specific to porting an application to 64-bit mode. These messages can help to narrow the scope of investigation or to pinpoint a particular coding construct.

The following table shows other options that can be helpful for detecting and correcting compile-time errors.

Other diagnostic options for compile-time errors

Option	Description
-qsrcmsg	Prints to standard error the source line that the compiler believes to contain the error, a line below it pointing to a particular spot in that source line, and the diagnostic message.
-qsource	Requests a compiler listing to be returned. The various sections of a listing include the source code with line numbers, the options specified, a listing of all files used in the compilation, a summary of the diagnostic messages by severity level, the number of lines read, and whether or not the compilation was successful. The attribute and cross-reference section of a listing can be produced by specifying the -qattr and -qxref options, respectively. The object section, which requires specifying the option -qlist , shows the pseudo assembly code generated by the compiler and is used for diagnosing execution time problems in cases where you suspect the program is not performing as expected due to code generation errors.
-qsuppress	Stops particular messages from being emitted by the compiler. You can suppress more than one message by listing the message numbers in a colon separated list.
-qflag	Stops specified diagnostic messages from being emitted to the terminal and the listing file. The option uses the single-letter severity codes of the default compiler message format to specify the level below which messages should be ignored.

32- and 64-bit application development

You can use XL C/C++ to develop both 32- and 64-bit applications. This section contains reference information and other portability considerations for moving C and C++ programs from 32- to 64-bit mode.

A porting exercise is a true port if the hardware and operating systems are the only things that are changed. Moving a 32-bit application to a 64-bit programming model as part of the process of migrating to AIX means that the exercise is no longer a true port but a development activity. A 32-bit application with any of the following characteristics is very likely to require changes when ported to a 64-bit environment:

- Reads and interprets kernel memory directly.
- Uses the `/proc` file system to access 64-bit processes.
- Uses a library that has only a 64-bit version.

- Is a device driver.
- Is being ported from a source platform that has interoperability issues with AIX.

Developing in 64-bit mode allows the application to take advantage of the newer, faster 64-bit hardware and operating systems to improve performance on large, complex, memory-intensive programs, such as database and scientific applications. Applications that are limited by a 32-bit address space, such as database applications, Web search engines, and scientific computing applications, are likely to benefit from a transition to the 64-bit mode. I/O bound applications in 64-bit mode can also realize improved performance by keeping data in memory rather than writing to disk, since disk I/O is usually slower than memory access.

The ability to handle larger problems directly in physical memory is perhaps the most significant performance benefit of 64-bit machines. However, some applications compiled in 32-bit mode perform better than when they are recompiled in 64-bit mode. Some reasons for this include:

- 64-bit programs are larger. The increase in program size places greater demands on physical memory.
- 64-bit long division is more time-consuming than 32-bit integer division.
- 64-bit programs that use 32-bit signed integers as array indexes might require additional instructions to perform sign extension each time the array is referenced.

Other disadvantages of 64-bit applications are that they require more stack space to hold the larger registers. Applications have a bigger cache footprint due to the larger pointer size. 64-bit applications do not run on 32-bit platforms.

Some ways to compensate for the performance liabilities of 64-bit programs are listed below.

- Avoid performing mixed 32- and 64-bit operations. For example, adding a 32-bit signed data type to a 64-bit data type requires the 32-bit type to be sign-extended to set the upper 32 bits of the register according to the sign. Setting the upper bits of the register slows the computation. If the 32-bit type were unsigned, then the upper bits of the register would be cleared.
- Avoid 64-bit long division whenever possible. Multiplication is usually faster than division. If you need to perform many divisions with the same divisor, assign the reciprocal of the divisor to a temporary variable, and change all divisions to multiplications with the temporary variable. For example, the function

```
double preTax(double total) { return total * (1.0 / 1.0825); }
```

will perform faster than the more straightforward:

```
double preTax(double total) { return total / 1.0825; }
```

The reason is that the division (1.0 / 1.0825) is evaluated once at compile time only, due to constant folding. This optimization is usually done by the compiler if the **-qnostrict** option is in effect (true when compiling at optimization levels **-O2** and higher).

- Use **long** variables as array indexes instead of signed, unsigned, and plain **int** types. Doing so frees the compiler from having to perform sign extension during array references.

Well-written code is likely to compile and run correctly with a minimum of rework and debugging if moved to the 64-bit programming model. The term *well-written*

implies conformance to the language standard and adherence to good programming practices. In the context of moving to a 64-bit programming model, the term also includes the notions that the code does not depend on a specific byte order nor on external data formats, and that it uses function prototypes, appropriate system header files, and system-derived data types

32- and 64-bit development environments on AIX

The C and C++ compilers and the AIX operating system offer two different programming models: ILP32 and LP64. ILP32, an acronym for integer/long/pointer 32, is the native 32-bit programming environment on AIX. The ILP32 data model provides a 32-bit address space, with a theoretical memory limit of 4 GB. LP64, an acronym for long/pointer 64, is the 64-bit programming environment on AIX and is the *de facto* standard data model on 64-bit UNIX-based systems from all major system vendors. Generally speaking, with the exception of data type size and alignments, LP64 supports the same programming features as the ILP32 model and is backward compatible with the most widely used `int` data type.

Compiler support

By default, the compiler invocations invoke the compiler and linker in 32-bit mode. The following features are provided to enable 64-bit development:

- `__64BIT__` preprocessor macro, which is predefined when compiling in 64-bit mode. The macro allows a programmer to select different lines of code for 32- and 64-bit execution in the same source file.
- `OBJECT_MODE` environment variable. This environment variable changes the default compilation mode to accept objects of either or both bit modes. A number of AIX utilities commonly employed in C and C++ application development use this environment variable. However, neither the compiler and nor the linker supports `OBJECT_MODE=32_64`, even though most C and C++ libraries provided by AIX are hybrid mode archives (both 32- and 64-bit objects are included).
- `-q64` compiler option. This option, which sets 64-bit mode support, works in conjunction with the `-qarch` option, which specifies the instruction set for the target architecture. `-q32` and `-q64` take precedence over the setting of the `-qarch` option. In conflicts between `-q32` and `-q64`, the last option specified prevails.
- `-qarch` support for 64-bit compilation. This option instructs the compiler to generate code for optimal execution on a given 64-bit architecture. Accordingly, only certain combinations of `-qarch` and `-q64` are accepted.

See *XL C/C++ Programming Guide* for more information.

AIX utility commands support

The AIX operating system provides a number of utility commands that deal with object files and that are necessary to manipulate the objects and libraries on AIX. These utilities support a 64-bit XCOFF object format when invoked with the `-X` option, which specifies the bit mode of the object files they process. The eXtended Common Object File Format, XCOFF, is the object format for AIX. The possible values for the `-X` are:

- `32` Process only 32-bit object files.
- `64` Process only 64-bit object files.
- `32_64` Process both 32- and 64-bit object files.

AIX utility commands for manipulating shared objects and libraries

Command	Description
ar	Maintains the indexed libraries used by the linkage editor. Silently ignores 64-bit objects when in 32-bit mode.
dump	Displays selected parts of an object file. In the context of linking and loading, the dump command is used to examine the header information of executable files and shared objects. The dump -H command enables you to determine the dependencies for an executable or shared object for symbol resolution at run time by displaying header information. The dump -Tv command displays the symbol definitions for a shared object or executable: the symbols the object is exporting at link time, the symbols the object or executable will try to import at load time, and, if known, the name of the shared object that contains those symbols.
file	Displays the bit mode of files and whether the file has been stripped.
genkld	Lists the shared objects that are loaded in the system memory, specifically in the system shared library segment. Private shared objects are not shown in the genkld command output, even if they are loaded into memory.
ldd	Lists the shared objects and archive members that will be loaded to start the executable.
lorder	Finds the best order for member files in an object library.
nm	Displays information about the symbols in object files, executables, and object file libraries. The nm -g command handles all archive members contained in the specified library archive. Unlike dump , nm does not display the shared object or archive member name that is expected to supply the symbol.
ranlib	Converts archive libraries to random libraries.
rtl_enable	Converts a shared library compiled for default linking, static linking, or lazy loading to one enabled for run-time linking.
size	Displays the section sizes of the XCOFF object files.
slibclean	Allows the root user only to unload all shared objects with a use count of zero from the system shared library segment. The utility is intended for use on production systems that are in a software maintenance phase, in particular, before the removal of applications no longer required or before updating installed applications.
strip	Reduces the size of an XCOFF object file by removing information used by the binder and symbolic debug information.

Related References

- *AIX 5L Version 5.2 Reference Documentation: Commands Reference*

Objects and libraries on AIX

Like other UNIX operating systems, the AIX operating system provides facilities for the creation, development, testing, and debugging of shared libraries and applications that use them. AIX shared library features are broadly compatible with other UNIX operating systems. However, AIX also provides facilities for the creation and use of *dynamically bound* shared libraries. With dynamic binding, external symbols referenced in user code and defined in a shared library are resolved by the loader at run time. Dynamic linking uses less memory to run programs and produces smaller executable files. In addition, AIX supports

dynamic loading, a programming scheme provided by a set of subroutines rather than by linker options or special object file types.

To implement and support this variety of linking methods, AIX requires files to be in the eXtensible Common Object File Format (XCOFF). When discussing objects and shared libraries, it is important to note the precise meanings of terms as they are used in the context of application development on AIX.

An *object file* is a generic term for a file containing executable code, data, relocation information, a symbol table, and other information. Multiple object files can be archived into a single *library archive file*, also referred to as simply a *library*. An object file that is contained in a library is referred to as an *archive member*. The advantage of a library over multiple object files is that fewer files need to be handled to create an executable.

To build an executable file on AIX, all participating object files and archive members must be in the same bit mode. However, a library archive can contain both 32- and 64-bit object modules. Most system libraries provided by AIX are hybrid mode. The AIX operating system provides command utilities for querying, maintaining, and manipulating objects and libraries, such as **ar** for creating and maintaining library archive files, **file** for displaying the bit mode of object files, **dump** for querying symbols in an archive, and **nm** for querying the available symbols in a library, given a particular bit mode.

Difference between a shared object and library on AIX

The terms *shared library* and *shared object* are often used interchangeably on other UNIX operating systems. What other UNIX operating systems refer to as *shared libraries* (also referred to as *dynamic link libraries* or *DLLs*) are properly referred to as shared objects on AIX.

There is a distinct difference between a shared library and a shared object on AIX. When a program is linked with several system libraries, the shared objects are *dependent modules* of the executable file. On AIX, the names of the shared objects do not have to be unique, except within the same library. Shared objects with the same name in different libraries are considered different modules. Most system libraries provided by AIX contain one or more shared objects.

A shared object is a single object file that has the SHROBJ flag in its XCOFF header. The naming convention for a shared object on AIX is *name.o*, which uses the default file name extension generated by compilers.

A shared library on AIX refers to an archive library file created by the **ar** command, in which one or more of the archive members is a shared object. The library can also contain nonshared object files (also referred to as *static* object files). The naming convention for a shared library on AIX is of the form *libname.a*, which is the file name extension expected by the linker.

XCOFF (eXtensible Common Object File Format) is the object file format for the AIX operating system. It is the formal definition of machine-image object and executable files. These object files are primarily used by the binder and the system loader. XCOFF extends the standard common object file format (COFF) to provide for dynamic linking and the replacement of units within an object file. XCOFF also provides for both 32-bit and 64-bit object and executable files.

Programs can be written to understand 32-bit XCOFF files, 64-bit XCOFF files, or both. The programs themselves may be compiled in either bit mode to create 32-bit or 64-bit programs. By defining preprocessor macros, applications can select the proper structure definitions from the XCOFF header files.

Assemblers and compilers produce XCOFF files as output. The binder combines individual object files into an XCOFF executable file. The system loader reads an XCOFF executable file to create an executable memory image of a program. The symbolic debugger reads an XCOFF executable file to provide access to functions and variables of an executable memory image.

Difference between shared and static objects on AIX

On many UNIX operating systems, the file name extension `.so` indicates a file for a shared object and the extension `.o`, one for a static object. On AIX, both static and shared object files use the file name extension `.o`. For shared object files, the extension `.o` is a convention rather than a requirement: the linker reads the file header to determine the content of the file. A shared object file is distinguished from a static object file in that a shared object file is fully linked and resolved, can be loaded and executed, and FLAGS field of the XCOFF file header has the SHROBJ bit set. The SHROBJ bit is set when the object is dynamically referenced. Otherwise, the linker treats the object as static.

AIX also supports shared objects with the file name `.so`. Apart from the conventions for the `.o` file name extension, a shared object file on AIX can be named using any extension. On AIX, the file name extension `.so` is sometimes used to indicate run-time linking.

The AIX operating system provides various AIX commands to display the XCOFF header of a file to determine whether an object is shared or static. The **dump -ov** command displays the XCOFF header of the specified file. The **dump -gov** command allows you to examine the archive members of the specified library to determine if archive members are shared or static.

Link time and load time

Shared objects and libraries are used in two stages when creating and executing a program on AIX.

At link time, the link editor searches the specified shared objects and libraries to resolve all undefined symbols needed to create the executable file. If a shared object contains a referenced symbol, the loader section in the XCOFF header of the resultant executable contains a reference to that shared object. The same is true for a referenced symbol in a library member that is shared object. Symbols are exported from those shared objects or libraries and imported to the executable file.

At program load time, the system loader reads the XCOFF header information in the executable and attempts to locate the referenced shared libraries. If all referenced shared objects and libraries are found, the executable is started. The system loader attempts to load the sections of program executable components into the appropriate segments in the process address space. The program text contained in shared objects and libraries is loaded into the global system memory by the first program that needs it and is subsequently shared by all programs that use it.

Diagnostics for link-time errors

Common link-time errors are unresolved symbols, duplicate symbols, and insufficient memory for the linker.

Unresolved symbols

Linker error warnings often arise when an application is linked with many libraries, particularly those supplied by a third party product. When an external symbol cannot be resolved, the link fails and an executable is not generated. The most common cause for unresolved symbol errors is missing input files. For example, if you call a library function that is not in the default C run-time library `libc.a`, you need to specify the archive library file in which the symbol is found.

The system link editor accepts input such as object files, shared object files, archive object files, libraries, and import and export files. How do you find the library file in which an unresolved symbol is defined? You can search the product documentation, you can include all supplied libraries in the link command and let the link editor find where the symbol is, or you might use the `nm` command to try find it yourself.

Certain linker options generate log files which can be analyzed to determine the library or object file that references the unresolved symbol. The log files can track interdependent or redundant libraries being used in error.

- The `-bnoquiet` option writes each binder subcommand and its results to stdout. Any unresolved symbol references appear in the output. The output also lists the symbols imported from specified library modules.
- The `-bmap:filename` option generates an address map. Unresolved symbols are listed at the top of the file, followed by imported symbols.
- The `-bloodmap:filename` option generates a log file that includes information on all of the arguments passed to the linker, the shared objects being read, and the number of symbols being imported. If an unresolved symbol is found, the log file lists the object file or shared object that references the symbol. To find an unresolved symbol in libraries supplied by a third-party product, you can use the log file to search the other libraries supplied by the product.

Duplicate symbols

Duplicate symbol errors usually indicate a programming error. It is incorrect to have multiple external function definitions of the same name. When source code contains multiple external function definitions, the link editor uses the first definition that it encounters, and the result may not be desirable. The recommended solution is to change the function name or to use a static function.

The use of template functions in C++ may also generate duplicate symbol errors at link time when the template is implicitly instantiated in multiple source files. The recommended solution is to use the `-qtemplateregistry` option and the template handling mechanism introduced with VisualAge C++ Professional, Version 6.

Another source of duplicate symbols is inlined functions. Every inlined function creates symbol table entries that appear as distinct instantiations. Compiling with the option `-qweaksymbol` on AIX 5.2 can reduce the number of these warnings.

Insufficient memory for the linker

Linking very large files can exhaust the memory allowed for the linker process. The solutions might be to increase the paging space or the resource limits for the user invoking the command. The **ulimit** command controls the resource limits for the user invoking the linker. Alternatively, you might consider running the linker process in the 32-bit large memory model.

Diagnostics for run-time errors

A program might compile and link successfully, yet produce unexpected results during execution. The compiler does not diagnose programming errors that do not violate the syntax of the language. This section describes some common errors, how to detect them, and how to correct them.

Uninitialized variables

An object of automatic storage duration is not implicitly initialized and its initial value is therefore indeterminate. If an **auto** variable is used before it is set, it may or may not produce the same results every time the program is run. The **-qinfo=gen** compiler option displays the location of **auto** variables that are used before being set. The **-qinitauto** option instructs the compiler to initialize all automatic variables to the specified value. This option reduces the run-time performance of the application and is recommended for debugging only.

Run-time checking

The **-qcheck** option inserts run-time checking code into the executable. The suboptions specify checking for null pointers, indexing outside of array bounds, and division by zero. Like **-qinitauto**, **-qcheck** degrades application performance and is recommended for debugging purposes only.

ANSI aliasing

Type-based aliasing, also referred to as ANSI aliasing, restricts the lvalues that can be safely used to access a data object. When compiling under a language level that enforces conformance to the language standards, the C and C++ compilers enforce type-based aliasing during optimization. The ANSI aliasing rule states that a pointer can only be dereferenced to an object of the same type or a compatible type. The exceptions are that sign and type qualifiers are not subject to type-based aliasing and that a character pointer can point to any type. The common coding practice of casting a pointer to an incompatible type and then dereferencing it violates this rule.

Turning off ANSI aliasing by setting **-qalias=noansi** may correct program behavior, but doing so reduces the opportunities for the compiler to optimize the application and thereby degrades run-time performance. The recommended solution is to change the program to conform to the type-based aliasing rule.

#pragma option_override

Sometimes an error appears only when optimization is used. It can be worthwhile, especially for complex applications, to turn off optimization for a function known to contain a programming error, while allowing the rest of the program to be optimized. The **#pragma option_override** directive allows you to specify alternate optimization options for specific functions.

The `#pragma option_override` directive can also be used to determine the function is causing the problem. The discovery is made by selectively turning off optimization for each function within the directive until the problem disappears.

Shared memory parallelization

Several proven techniques are available to achieve parallel execution of a program and more rapid job completion than running on a single processor. These techniques include:

- Directive-based shared memory parallelization (SMP)
- Instructing the compiler to automatically generate shared memory parallelization
- Message passing based shared or distributed memory parallelization (MPI)
- POSIX threads (pthreads) parallelization
- Low-level UNIX parallelization using `fork()` and `exec()`

The portability requirements for the application are definitely a factor in selecting the best technique to use. The choice is also highly dependent on the application, the programmer's skills and preferences, and the characteristics of the target machine. The two principal ways of accomplishing parallelization on AIX is by using hand-coded POSIX threads (pthreads) and OpenMP directives.

The parallel programming facilities of the AIX operating system are based on the concept of threads. Parallel programming exploits the advantages of multiprocessor systems, while maintaining a full binary compatibility with existing uniprocessor systems. This means that a multithreaded program that works on a uniprocessor system can take advantage of a multiprocessor system without recompiling.

Pthreads offer great flexibility for making effective use of multiple processors because they provide the maximal amount of control of the parallelization process. The trade-off is a considerable increase in code complexity. In many cases, the simpler approach of using OpenMP directives, SMP-enabled libraries, or the automatic SMP capabilities of compilers is preferable. The explicit use of threads does not necessarily lead to better performance. Debugging multithreaded applications is awkward at best. However, in some programs, it is the only viable approach.

The following lists some pros and cons of the different techniques.

Automatic parallelization by the compiler

- Easy to implement (compile with `-qsmp=auto`).
- Enables teamwork easily.
- Limited scalability because data scoping is neglected.
- Compiler-dependent (even on the release of a particular compiler).
- Not necessarily portable.

SMP-enabled libraries

- Requires the least effort.
- Not necessarily portable (usually is proprietary).
- Limited flexibility.

OpenMP directives

- Portable.
- Potentially better scalability of the automatic parallelization.

- Uniform memory access is assumed.

Hybrid approach (subset or mixture of OpenMP and pthreads, or UNIX fork() and exec() parallelization, platform-specific constructs)

- Might enable teamwork.
- Needs a well-tested concept to assure performance and portability.
- Not necessarily portable.

pthreads

- Portable.
- Provides maximal control over the parallelization process.
- Potentially the best scalability of the automatic parallelization.
- Needs experienced programmers to handle code complexity.

OpenMP directives

OpenMP directives are a set of commands that instruct the compiler how a particular loop should be parallelized. The existence of the directives in the source removes the need for the compiler to perform any parallel analysis on the parallel code. The use of OpenMP directives requires the presence of the pthread libraries to provide the necessary infrastructure for parallelization.

The OpenMP directives address three important issues of parallelizing an application. First, clauses and directives are available for scoping variables. Frequently, variables should not be shared; that is, each processor should have its own copy of the variable. Second, work sharing directives specify how the work contained in a parallel region of code should be distributed across the SMP processors. Finally, there are directives for synchronization between the processors.

The compiler supports the OpenMP Version 2.0 specification.

Related References

- Appendix B, “OpenMP compliance and support,” on page 67

Threads on AIX

The implementation of threads on AIX is classified as an external portability issue because it deals with external interfaces and their assumed semantics that the program uses. The AIX threads library conforms to the Single UNIX Specification Version 2, which includes the IEEE POSIX 1003.1c standard, known as the POSIX thread standard. The library also conforms to the Open Group 98 specification, which adds extended thread functions to the POSIX thread standard. The standardized interface of pthreads therefore assures the portability of a threaded program.

The POSIX threads library on AIX

On AIX, POSIX threads (pthreads) are defined as a set of C language programming types and subroutine calls, implemented with a header file (/usr/include/pthread.h) and the POSIX thread library (/usr/lib/libpthreads.a).

The following data types are defined for the threads library in the pthread.h header file. The definition of these data types can vary between systems.

pthread_t	Identifies a thread.
pthread_attr_t	Identifies a thread attributes object.

<code>pthread_cond_t</code>	Identifies a condition variable.
<code>pthread_condattr_t</code>	Identifies a condition attributes object.
<code>pthread_key_t</code>	Identifies a thread-specific data key.
<code>pthread_mutex_t</code>	Identifies a mutex.
<code>pthread_mutexattr_t</code>	Identifies a mutex attributes object.
<code>pthread_once_t</code>	Identifies a one-time initialization object.

AIX provides binary compatibility for existing multithreaded applications that were written for the draft of Version 7 of the POSIX threads standard. The compatibility POSIX thread library, `/usr/lib/libpthreads_compat.a`, is only provided for backward compatibility for those applications. The compatibility POSIX thread library supports 32-bit applications only.

The following operating system files provide the AIX implementation of pthreads:

<code>/usr/include/pthread.h</code>	C/C++ header with most pthread definitions.
<code>/usr/include/sched.h</code>	C/C++ header with some scheduling definitions.
<code>/usr/include/unistd.h</code>	C/C++ header with pthread_atfork() definition.
<code>/usr/include/sys/limits.h</code>	C/C++ header with some pthread definitions.
<code>/usr/include/sys/pthdebug.h</code>	C/C++ header with most pthread debug definitions.
<code>/usr/include/sys/sched.h</code>	C/C++ header with some scheduling definitions.
<code>/usr/include/sys/signal.h</code>	C/C++ header with pthread_kill() and pthread_sigmask() definitions.
<code>/usr/include/sys/types.h</code>	C/C++ header with some pthread definitions.
<code>/usr/lib/libpthreads.a</code>	32-bit/64-bit library providing UNIX98 and POSIX 1003.1c pthreads.
<code>/usr/lib/libpthreads_compat.a</code>	32-bit only library providing POSIX 1003.1c Draft 7 pthreads.
<code>/usr/lib/profiled/libpthreads.a</code>	Profiled 32-bit/64-bit library providing UNIX98 and POSIX 1003.1c pthreads.
<code>/usr/lib/profiled/libpthreads_compat.a</code>	Profiled 32-bit only library providing POSIX 1003.1c Draft 7 pthreads.

Thread-unsafe libraries may be used by only one thread in a program. Thread-unsafe code can only be used safely by the main thread. This restriction is due to access of the global `errno`, rather than the thread-specific `errno`, and also applies to library calls made by the main thread.

Pthreads on AIX

Traditionally, multiple single-threaded processes have been used to achieve parallelism, but some programs can benefit from a finer level of parallelism. Multithreaded processes offer parallelism within a process and share many of the concepts involved in programming multiple single-threaded processes.

This section discusses points about the implementation of parallel programming facilities in AIX, as compared to that in other UNIX systems.

Terminology: In traditional single-threaded process systems, thread and process characteristics are grouped into a single entity called a *process*. In other systems, a *thread* is sometimes synonymous with a *lightweight process*, or the meaning of the word *thread* is sometimes only slightly different from *process*.

On AIX, a distinction is made between a *kernel thread* and a *user thread*. On AIX, the term *lightweight process* usually refers to a kernel thread. A *user thread* is an independent flow of control that operates within the same address space as other independent flows of controls within a process. In the remainder of this discussion, the term *thread* refers to a user thread.

A *kernel thread* is the schedulable entity handled by the system scheduler. On AIX, a kernel thread runs within a process, but can be referenced by any other thread in the system. However, a kernel thread cannot be directly controlled by the programmer. Kernel threads are strongly implementation-dependent, and so libraries, such as the POSIX threads library, provide user threads to facilitate writing portable programs.

A *user thread* is an entity used by programmers to handle multiple flows of controls within a program. The standardized API for handling user threads is provided by the threads library. A user thread only exists within a process and cannot reference a user thread in another process. The library uses a proprietary interface to handle kernel threads for executing user threads.

Properties: In traditional single-threaded process systems, a process has a set of properties. In a multithreaded process system like AIX, these properties are divided between processes and threads.

A *process* in a multithreaded system is the changeable entity. It must be considered as an execution frame. It has traditional process attributes, such as process ID, process group ID, user ID, and group ID, an environment, a working directory. A process also provides a common address space and common system resources, such as file descriptors, signal actions, shared libraries, and interprocess communication tools.

A *thread* is the schedulable entity, with only those properties that are required to ensure its independent control of flow. These properties include stack, scheduling policy or priority, set of pending and blocked signals, thread-specific data. An example of thread-specific data is the **errno** error indicator. On AIX, each thread has its own **errno** error indicator. In multithreaded systems, **errno** is usually a subroutine returning a thread-specific **errno** value rather than a global variable. Other systems may provide other implementations of **errno**.

Threads within a process must not be considered as a group of processes. All threads share the same address space. This means that two pointers having the same value in two threads refer to the same data. Also, if any thread changes one of the shared system resources, all threads within the process are affected. For example, if a thread closes a file, the file is closed for all threads.

A thread does not maintain a list of created threads, nor does it know the thread that created it. Thread creation differs from process creation in that no parent-child

relation exists between threads: all threads, except the initial thread, are on the same hierarchical level. The initial thread is automatically created by the operating system when a process is created.

A thread attributes object, which encapsulates the attributes of the thread, must be defined before the thread is created. An entry-point routine and an argument must also be specified at time of creation. Every thread has an entry-point routine with one argument, but the same entry-point routine may be used by several threads.

Thread models and virtual processors: User threads are mapped to kernel threads by virtual processors (VPs) in the threads library. The way the mapping is done is called the *thread model*. The default thread model on AIX is the M:N model, in which all user threads are mapped to a pool of kernel threads; all user threads run on a pool of virtual processors. This is the most efficient and complex thread model, in which the facilities for user threads programming are shared between the threads library and kernel threads. The M:N model can also accommodate the case in which a single user thread is bound to a specific virtual processor (the 1:1 thread model) and all unbound user threads share the remaining VPs.

The 1:1 thread model maps each user thread to one kernel thread; all user threads run on one VP. Most of the facilities for user threads programming are directly handled by the kernel threads. The 1:1 model can be enabled by setting the value of the `AIXTHREAD_SCOPE` variable to `S`.

The M:1 thread model can be used on any system, especially on traditional single-threaded systems. All user threads are mapped to one kernel thread by a library scheduler; all user threads run on one VP. All facilities for user threads programming are completely handled by the library.

Synopsis of the pthread life cycle: Each thread in a threaded program has its own private program counter, stack, and registers. The memory state and file descriptors are shared.

The pthread.h header file

The `pthread.h` header file must be the first included file of each source file using the threads library to ensure that thread-safe subroutines are used. The header file contains all subroutine prototypes, macros, and other definitions for using the threads library. It also redefines the `errno` global variable as a function returning a thread-specific `errno` value. The `errno` identifier is, therefore, no longer an lvalue in a multithreaded program.

The following global symbols are defined in the `pthread.h` file:

<code>POSIX_REENTRANT_FUNCTIONS</code>	Specifies that all functions should be reentrant.
<code>POSIX_THREADS</code>	Specifies the availability of the POSIX threads API.

Compiler invocation

A threaded application should be compiled and linked with one of the `_r`-suffixed invocations of the compiler. These are the reentrant invocation commands, which ensure that adequate and appropriate options are used and that the program is

linked with the reentrant and thread-safe libraries. For example `xlcr` defines the symbol `_THREAD_SAFE` and links with the pthreads library.

Thread creation

The execution of a pthread program begins as a single thread created by the operating system. Additional threads are created and terminated as necessary to concurrently schedule work onto the available processors.

Threads, except for the initial thread, are created using the `pthread_create` function. This function has four arguments: A thread identifier, which is returned upon successful completion, a pointer to a thread-attributes object, the function that the thread will execute, and the argument to be passed to the thread function. The thread function takes a single pointer argument (of type `void *`) and returns a pointer (of type `void *`). In practice, the argument to the thread function is often a pointer to a structure, and the structure may contain many data items that are accessible to the thread function.

A program can create a fixed number of threads. However, in many cases, it can be useful to have the program decide how many threads to create at run time while providing the ability to override the default behavior by setting an environment variable. For example, for OpenMP programs the default is to create as many threads as processors are available. In AIX, you can get the number of online processors by calling the `sysconf` routine from `libc`.

Thread termination

A thread terminates implicitly when the execution of the thread function is completed. A thread can terminate itself explicitly by calling `pthread_exit`. It is also possible for one thread to terminate other threads by calling the `pthread_cancel` function.

The initial thread has a special property. If the initial thread reaches the end of its execution stream and returns, the exit routine is invoked, and, at that time, all threads that belong to the process will be terminated. However, the initial thread can create detached threads, and then safely call `pthread_exit`. In this case, the remaining threads will continue execution of their thread functions and the process will remain active until the last thread exits. In many applications, it is useful for the initial thread to create a group of threads and then wait for them to terminate before continuing or exiting. This can be achieved with threads that are joinable. On AIX, the default setting is detached. The function `pthread_join` suspends the calling thread until the referenced thread has terminated. The system scope attribute of the `AIXTHREAD_SCOPE` environment variable is appropriate when N threads are supposed to run on N processors concurrently.

Synchronization: In multithreaded programs, the same functions and the same resources may be accessed concurrently by several flows of control. To protect resource integrity, code written for multithreaded programs must be both reentrant and thread-safe. Reentrance and thread safety are separate concepts that can pertain to a function. A function can be either reentrant, thread-safe, both, or neither.

A *reentrant function* does not hold static data over successive calls, nor does it return a pointer to static data. All data is provided by the caller of the function. A reentrant function must also not call a non-reentrant function. In most cases, a

non-reentrant function will need to be replaced with a function with a modified interface in order for the function to become reentrant. Non-reentrant functions are usually thread-unsafe, as well.

A *thread-safe function* protects shared resources from concurrent access by locks. Thread safety concerns only the implementation of a function and does not affect its external interface. In multithreaded programs, all functions called by multiple threads must be thread-safe.

The use of global data is thread-unsafe without explicit synchronization or serialization. Global data should be maintained per thread or encapsulated so that its access can be serialized. A thread may read an error code corresponding to an error caused by another thread. In AIX, each thread has its own `errno` value.

Some of the standard C subroutines are non-reentrant, such as the `ctime` and `strtok` subroutines, even though they belong to libraries are considered thread-safe. Reentrant versions of subroutines have the name of the original subroutine with the suffix `_r`.

The distinction between threadprivate and shared variables is essential for the correctness and performance of a program. This is also true for programs that achieve shared memory parallelization using OpenMP directives. The access to shared variables must be synchronized to avoid conflicts and to assure correct results. However, the use of synchronization needs to be balanced with its degradation of performance and scalability.

A major difficulty of parallel programming for shared memory is to find the right balance of local and global variables, since the scoping defines which variables are private or shared. Contention for global variables, as in a reduction sum, is a major source of performance problems. The introduction of temporary local variables often helps to resolve such problems.

In multithreaded applications, the update of shared memory locations is usually protected with mutex locks. The operating system ensures that access to the shared data is serialized. At a given time, only one thread can enter the region between lock and unlock to modify the data.

Sharing memory between AIX processes: AIX and most UNIX systems allow several processes to share a common data space, known as shared memory. The process-sharing attributes for condition variables and mutexes are meant to allow these objects to be allocated in shared memory to support synchronization among threads belonging to different processes. However, because there is no industry-standard interface for shared memory management, the process-sharing POSIX option is not implemented in the AIX threads library.

Debugging a multithreaded program

The AIX operating system provides the `dbx` command, which invokes a symbolic debug program for C/C++ and Fortran programs. The `dbx` command has subcommands for displaying thread-related objects, including attribute, condition, mutex, and thread.

For kernel programming, the operating system provides a kernel debug program. For more information, see *AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts*, which is part of the AIX documentation.

Tuning a multithreaded program

One aspect of tuning a multithreaded program concerns adjusting the amounts of time spent in different states in the thread life cycle. Various threading functions control the pace and resource consumption of the sequence of life cycle events. After parallel work for a thread has completed, the thread waits (spins) for a period (the spin wait time), but it consumes processor time while waiting. After the spin wait time and if a yield wait time has been specified, the thread can yield its place on the kernel thread to another runnable thread. If the yield wait time has expired, the thread enters a sleep state. Reactivating a thread from a sleep state is more resource-intensive than if the thread is in a yielded state.

The AIX operating system provides environmental variables that modify the behavior of various threading functions. The following table presents selected environment variables that can be useful when tuning a multithreaded program.

Selected environment variables that affect threading

Description	Syntax
<p>AIXTHREAD_SCOPE</p> <p>The thread contention scope controls the mapping of application-level pthreads to entries in the system scheduling queue. AIXTHREAD_SCOPE permits exploring application behavior by being able to change the thread scope without having to modify the application. A single process may contain pthreads of both scopes. This is achieved when the attribute argument to pthread_create() is not NULL. Then, the contents of the attribute structure determines the scope of the thread.</p>	<p>►►—AIXTHREAD_SCOPE=$\begin{matrix} \text{P} \\ \text{S} \end{matrix}$—►►</p> <p>where</p> <p>P Represents process contention scope (default), which implies the M:N thread model. Best used when there are many more threads than processors.</p> <p>S Represents system contention scope, which implies the 1:1 thread model. Each user thread is directly mapped to one kernel thread.</p>
<p>AIXTHREAD_MNRATIO</p> <p>Controls the scaling factor used within the pthread library when creating and terminating pthreads. Applies only to process-scope threads. The default M:N ratio is 8:1, that is, eight pthreads for every kernel thread. Modify the M:N ratio by setting and exporting the environmental variable:</p>	<p>►►—AIXTHREAD_MNRATIO=$M:N$—►►</p> <p>where</p> <p>M Represents the number of user threads. Default value is 8.</p> <p>N Represents the number of kernel threads. Default value is 1.</p>
<p>SPINLOOPTIME</p> <p>Controls the number of times a pthread will attempt to obtain a mutex before blocking on the mutex availability. Modify the spin loop time by setting and exporting the environmental variable:</p>	<p>►►—SPINLOOPTIME=N—►►</p> <p>where</p> <p>N Represents the number of times to retry a busy lock before yielding to another pthread. Default value is 40.</p>
<p>YIELDLOOPTIME</p> <p>Controls the number of times the system yields the processor when trying to acquire a busy spin lock before going to sleep. If the pthread spins on the lock and cannot get it, the thread will be put to sleep. Increasing the value of yield loop time prevents the pthread delays the pthread from entering sleep state. This variable is used only if SPINLOOPTIME is also set. The value of yield loop time can be modified by setting and exporting the environmental variable:</p>	<p>►►—YIELDLOOPTIME=N—►►</p> <p>where</p> <p>N Represents the number of times to yield to acquire a busy mutex or spin lock. Default value is 0.</p>

Selected environment variables that affect threading

Description	Syntax
<p>AIXTHREAD_MINKTHREADS</p> <p>Sets the minimum number of kernel threads that should be used for a process. An average threaded process will have at least this number of kernel threads available for scheduling pthreads.</p>	<p>▶▶—AIXTHREAD_MINKTHREADS=<i>N</i>—▶▶</p> <p>where</p> <p><i>N</i> Represents the number of kernel threads. Default value is 8.</p>
<p>AIXTHREAD_MUTEX_DEBUG</p> <p>Switch that controls the collection of debug information regarding mutexes in running thread processes. Default is OFF. Maintaining the debug information may adversely affect performance.</p>	<p>▶▶—AIXTHREAD_MUTEX_DEBUG=$\begin{cases} \text{OFF} \\ \text{ON} \end{cases}$—▶▶</p>
<p>AIXTHREAD_COND_DEBUG</p> <p>Switch that controls the collection of debug information regarding condition variables. Default is OFF. Maintaining the debug information may adversely affect performance.</p>	<p>▶▶—AIXTHREAD_COND_DEBUG=$\begin{cases} \text{OFF} \\ \text{ON} \end{cases}$—▶▶</p>
<p>AIXTHREAD_RWLOCK_DEBUG</p> <p>Causes the pthreads library to maintain a list of all read-write locks that can be viewed by debugging programs. Default is OFF.</p>	<p>▶▶—AIXTHREAD_RWLOCK_DEBUG=$\begin{cases} \text{OFF} \\ \text{ON} \end{cases}$—▶▶</p>
<p>AIXTHREAD_GUARDPAGES</p> <p>Specifies the number of 4 KB guard pages to create. Guard pages are used to detect when a thread stack has grown beyond its maximum size and to guard against errant memory writes. The thread stack, created on the process heap, can be protected by setting read-only guard pages at the top of the stack. Any attempt to write onto these pages results in an immediate segmentation violation. Investigating the conditions at the time of the stack overflow can help to debug the program and determine an appropriate corrective action. Set and export this variable.</p>	<p>▶▶—AIXTHREAD_GUARDPAGES=<i>N</i>—▶▶</p> <p>where</p> <p><i>N</i> Represents the number of 4 KB size pages. Default value is 0.</p> <p>If the application specifies its own stack or uses large pages for its process heap, no guard pages are created.</p>
<p>AIXTHREAD_SLPRATIO</p> <p>Sets the sleep ratio for the system. The value tells the system how many kernel threads should be held in reserve for sleeping pthreads. This tuning parameter allows greater management of kernel resources. The default sleep ratio is 1:12.</p>	<p>▶▶—AIXTHREAD_SLPRATIO=<i>k</i>:<i>p</i>—▶▶</p> <p>where</p> <p><i>k</i> Represents the number of kernel threads. Default value is 1.</p> <p><i>p</i> Represents the number of sleeping pthreads. Default value is 12.</p> <p>Any positive integer value may be specified for <i>k</i> and <i>p</i>. If <i>k</i> > <i>p</i>, then the ratio is treated as 1:1 (that is, you cannot specify more kernel threads than pthreads).</p>

Selected environment variables that affect threading

Description	Syntax
<p>AIXTHREAD_STK</p> <p>Modifies the thread stack size. Use this environment variable to specify stacks of up to 256MB in size instead of adjusting a parameter in the pthread attribute structure and then recompiling and rebuilding the application. Note that increasing the size of the thread stack decreases the amount of space available for dynamically allocated memory because each thread stack is created on the process heap. Export this environment variable to modify the stack size for pthreads created without specifying the stack size programmatically.</p>	<p>▶—AIXTHREAD_STK=<i>N</i>—▶</p> <p>where</p> <p><i>N</i> Represents the number of bytes. The default thread stack size is 96 KB for 32-bit applications and 192 KB for 64-bit applications.</p>

Features related to GNU C and C++ portability

To facilitate porting an application or code developed with GNU C, XL C/C++ supports a subset of the GNU C and C++ language extensions to C99 and Standard C++. The tables in this section list the features that are supported, unsupported, and those for which the syntax is accepted but the semantics ignored.

To use *supported* extensions with your C code, use the `xlc` or `cc` invocation commands, or specify one of `-q|langlvl=extc89`, `-q|langlvl=extc99`, or `-q|langlvl=extended`. To use these features with your C++ code, specify the `-q|langlvl=extended` option. In C++, all supported GNU C and C++ features are accepted by default.

In the following tables, extensions marked *accept/ignore* are recognized by the compiler as acceptable programming keywords, but the GNU C/C++ semantics are not supported. This means that compilation does not halt if the compiler encounters an *accept/ignore* keyword or extension, but the GNU semantics are not implemented in the application. Compiling source code that uses these extensions under a strict language level (`stdc89`, `stdc99`) will result in error messages.

Related References

The GNU C and C++ language extensions are fully documented in the GNU manuals at <http://gcc.gnu.org/onlinedocs>.

Function attributes

Use the keyword `__attribute__` to specify special attributes when making a function declaration or definition. This keyword is followed by an attribute specification inside double parentheses. XL C/C++ supports a subset of the GNU C and C++ function attributes. Behavior described as *accept/ignore* means that the syntax is accepted, but the semantics are ignored, and compilation continues.

GNU C/C++ function attribute compatibility with XL C/C++

Function Attribute	Behavior
<code>alias</code>	supported
<code>always_inline</code>	supported
<code>cdecl</code>	accept/ignore

GNU C/C++ function attribute compatibility with XL C/C++

Function Attribute	Behavior
const	supported
constructor	accept/ignore
destructor	accept/ignore
dlllexport	accept/ignore
dllimport	accept/ignore
eightbit_data	accept/ignore
exception	accept/ignore
format	supported
format_arg	supported
function_vector	accept/ignore
interrupt	accept/ignore
interrupt_handler	accept/ignore
longcall	accept/ignore
model	accept/ignore
no_check_memory_usage	accept/ignore
no_instrument_function	accept/ignore
noinline	supported
noreturn	supported
pure	supported
regparm	accept/ignore
section	accept/ignore
stdcall	accept/ignore
tiny_data	accept/ignore
weak	supported


Related References

- "Function attributes" in *XL C/C++ Language Reference*

Variable attributes

Use the keyword `__attribute__` to specify special attributes of variables or structure fields. This keyword is followed by an attribute specification inside double parentheses. XL C/C++ supports a subset of the GNU C and C++ variable attributes. Behavior described as *accept/ignore* means that the syntax is accepted, but the semantics are ignored, and compilation continues.

GNU C/C++ variable attribute compatibility with XL C/C++

Variable Attribute	Behavior
aligned	supported
 init_priority	accept/ignore
mode	supported
model	accept/ignore
nocommon	accept/ignore

GNU C/C++ variable attribute compatibility with XL C/C++

Variable Attribute	Behavior
packed	supported
section	accept/ignore
transparent_union	accept/ignore
unused	accept/ignore
weak	accept/ignore

Related References

- "Variable attributes" in *XL C/C++ Language Reference*

Type attributes

Use the keyword `__attribute__` to specify special attributes of struct and union types when you define these types. This keyword is followed by an attribute specification inside double parentheses. XL C/C++ supports a subset of the GNU C and C++ type attributes. Behavior described as *accept/ignore* means that the syntax is accepted, but the semantics are ignored, and compilation continues.

GNU C/C++ type attribute compatibility with XL C/C++

Type Attribute	Behavior
aligned	supported
packed	supported
transparent_union	▶ C supported
unused	accept/ignore

Related References

- "Type attributes" in *XL C/C++ Language Reference*

GNU C and C++ assertions



Use *assertions* to test what sort of computer or system the compiled program will run on. The assertions `#cpu`, `#machine`, and `#system` are predefined. You can also define assertions with the preprocessing directives `#assert` and `#unassert`.

GNU C and C++ assertions in XL C/C++

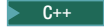
GNU C Assertions	Behavior
<code>#assert</code>	supported
<code>#unassert</code>	supported
<code>#cpu</code>	supported
<code>#machine</code>	supported
<code>#system</code>	supported possible values are aix and unix

Other extensions related to GNU C and C++

The following features related to GNU C and C++ are supported under extended language levels (`extc89`, `extc99`, `extended`).

- Use directive `#warning` to cause the preprocessor to issue a warning and continue processing.
- Use directive `#include_next` to specify inclusion of the next header file in a directory after the current one.
- Local labels can be declared at the start of each lexical block.
- Use a brace-enclosed compound statement inside of parentheses as an expression.
- Refer to the type of an expression with the `__typeof__` keyword.
- Use compound expressions, conditional expressions, and casts as lvalues.
- Use a computed `goto` statement to jump to a label, which has had its address taken and the address is used as a value.
- Use keyword `__alignof__` to inquire about variable alignment, or the alignment usually required by a type.
- Use alternate spelling of these keywords: `__asm__`, `__const__`, `__volatile__`, `__inline__`, `__signed__`, and `__typeof__`.
- Use the `__extension__` keyword to avoid an error when using an extended language feature in a strict language level mode.
- An array of zero length can occur without generating an error.
-  A function definition that appears within the definition of another function (a nested function) is permitted.
-  A union member can be cast to the union type to which it belongs.

Under extended language levels (`extc89`, `extc99`, `extended`), XL C/C++ recognizes the syntax of the following features, but their semantics are not supported.

-  The declaration of a register variable, either global or local, can suggest a preferred register.

Appendix A. Language support

This appendix discusses the implementations of the C and C++ programming languages and the language extensions provided by XL C/C++.

Compatibility with ISO/IEC International Standards

XL C/C++ can foster a programming style that emphasizes portability. Syntax and semantics constitute a complete specification of a programming language, but conforming implementations of a particular language specification can differ due to language extensions.

A program that conforms strictly to its language specification will have maximum portability among different environments. In theory, a program that compiles correctly with one standards-conforming compiler and that does not use any extension or implementation-defined behavior, will compile and execute properly under all other conforming compilers, insofar as hardware differences permit. A program that correctly exploits the extensions to the language that are provided by the language implementation can improve the efficiency of its object code.

ISO/IEC 14882:2003(E) International Standard compatibility


XL C/C++ is consistent with the ISO/IEC International Standard 14882:2003(E), which specifies the form and establishes the interpretation of programs written in the C++ programming language. The international standard is designed to promote the portability of C++ programs among a variety of implementations. ISO/IEC 14882:1998 was the first C++ language.

ISO/IEC 9899:1990 International Standard compatibility

The ISO/IEC 9899:1990 International Standard (also known as C89) specifies the form and establishes the interpretation of programs written in the C programming language. This specification is designed to promote the portability of C programs among a variety of implementations. This Standard was amended and corrected by ISO/IEC 9899/COR1:1994, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996. To ensure that your source code adheres strictly to the amended and corrected C89 standard, specify the `-qlanglvl=stdc89` compiler option.

ISO/IEC 9899:1999 International Standard support

The ISO/IEC 9899:1999 International Standard (also known as C99) is an updated standard for programs written in the C programming language. It is designed to enhance the capability of the C language, provide clarifications to C89, and incorporate technical corrections. XL C/C++ supports many features of this language specification.

 The C compiler supports all language features specified in the C99 Standard. To ensure that your source code adheres to this set of language features, use the `c99` invocation command. Note that the Standard also specifies features in the run-time library. These features may not be supported in the current run-time library and operating environment. The availability of system header files provides an indication of whether such support exists.

Major features in C99

XL C/C++ implements all C99 language features. The following is a table of selected major features. The references refer to articles in *XL C/C++ Language Reference*.

ISO/IEC 9899:1999 international standard extensions to IBM C

C99 Feature	Related Reference
restrict type qualifier for pointers	The restrict Type Qualifier
universal character names	The Unicode Standard
predefined identifier <code>__func__</code>	Predefined Identifiers
function-like macros with variable and empty arguments	Function-Like Macros
<code>_Pragma</code> unary operator	The <code>_Pragma</code> Operator
variable length array	Variable Length Arrays
static keyword in array index declaration	Arrays
complex data type	Complex Types
long long int and unsigned long long int types	Integer Variables
hexadecimal floating-point constants	Hexadecimal Floating Constants
compound literals for aggregate types	Compound Literals
designated initializers	Initializers
C++ style comments	Comments
implicit function declaration not permitted	Function Declarations
mixed declarations and code	The for Statement
<code>_Bool</code> type	Simple Type Specifiers
inline function declarations	Inline Functions
initializers for aggregates	Initializing Arrays Using Designated Initializers

Changes and clarifications of C89 supported in C99

Certain specifications in the C99 Standard are based on changes and clarifications of the C89 standard, rather than on new features of the language. XL C/C++ supports all C99 language features, including the following:

- Flexible array members are allowed. The last member of a structure with two or more members can be declared without the size.
- Declaring implicit int is not supported. All declarations must have a type specifier.
- Trailing commas are allowed in enumeration specifiers.
- Duplicate type qualifiers are accepted and ignored, unless explicitly specified otherwise.
- A diagnostic message will be issued if a required expression is missing from the return statement.
- Constant expressions evaluated during preprocessing now use long long and unsigned long long data types.
- Empty macro arguments are allowed in function-like macros.
- The maximum value of `#line` has increased to 2 147 483 647.

C99 features in XL C/C++

Some features of the ISO/IEC 9899:1999 International Standard (C99) are also implemented in C++. These extensions are available under the `-qlanglvl=extended` compiler option.

ISO/IEC 9899:1999 international standard extensions to IBM C++

C99 Feature	Reference
restrict type qualifier for pointers	The restrict Type Qualifier
universal character names	The Unicode Standard
predefined identifier <code>__func__</code>	Predefined Identifiers
variable length array	Variable Length Arrays
complex data type	Complex Types
hexadecimal floating-point constants	Hexadecimal Floating Constants
compound literals for aggregate types	Compound Literals
function-like macros with variable and empty arguments	Function-Like Macros
<code>_Pragma</code> unary operator	The <code>_Pragma</code> Operator

Enhanced language level support

The `-qlanglvl` compiler option is used to specify the supported language level, and therefore affects the way your code is compiled. You can also specify the language level implicitly by using different compiler invocation commands. In general, a valid program that compiles and runs correctly under a standard language level should continue to compile correctly and run to produce the same result with the orthogonal extensions enabled.

For example, to compile C programs so that they comply strictly with the ISO/IEC 9899:1990 International Standard (C89), you need to specify `-qlanglvl=stdc89`. The `stdc89` suboption instructs the compiler to strictly enforce the standard, and not to allow any language extensions. (The `c89` compiler invocation command specifies this language level implicitly.)

You can also use extensions to the standard language levels. Extensions that do not interfere with the standard features are called *orthogonal* extensions. For example, when you compile C programs, you can enable extensions that are orthogonal to C89 by specifying `-qlanglvl=extc89`.








Most of the language features described in the ISO/IEC 9899:1999 International Standard (C99) are considered orthogonal extensions to C89.

When you compile C++ programs, you can enable the use of orthogonal extensions by specifying `-qlanglvl=extended`.

Non-orthogonal extensions, on the other hand, can interfere or conflict with aspects of the language as described in one of the international standards. Acceptance of these extensions must be explicitly enabled by a particular compiler option. Reliance on non-orthogonal extensions reduces the ease with which your application can be ported to different environments.

The main suboptions for the `-q|ang|vl` option are listed below.

Selected `-q|ang|vl` suboptions

-q ang vl Suboption	Suboption Description
<code>-q ang vl=stdc99</code>	 Specifies strict conformance to the C99 standard.
<code>-q ang vl=stdc89</code>	 Specifies strict conformance to the C89 standard.
<code>-q ang vl=ansi</code>	 Specifies strict conformance to the C89 standard and enables the <code>-q ong ong</code> compiler option.
<code>-q ang vl=extc99</code>	 Enables all extensions orthogonal to C99.
<code>-q ang vl=extc89</code>	 Enables all extensions orthogonal to C89.
<code>-q ang vl=extended</code>	 Enables all extensions orthogonal to C89 and specifies the <code>-q up conv</code> compiler option.  Enables all the orthogonal extensions on top of Standard C++.

Appendix B. OpenMP compliance and support

The OpenMP Application Program Interface (API) is a portable, scalable programming model that provides a standard interface for developing multiplatform, shared-memory parallel applications in C, C++, and Fortran. The specification is defined by the OpenMP organization, a group of major computer hardware and software vendors, which includes IBM.

XL C/C++ is compliant with OpenMP Specification 2.0. The compiler recognizes and preserves the semantics of the following OpenMP V2.0 elements:

- Comma delimiter for multiple clauses in the `#pragma omp` directive.
- The `num_threads` clause.
- The `copyprivate` clause.
- `threadprivate` static block scope variables.
- Support for C99 variable length arrays.
- Redundant declaration of private variables.
- Timing routines `omp_get_wtick` and `omp_get_wtime`.

The directives, library functions, and environment variables described below allow you to create and manage parallel programs while maintaining portability.

To enable OpenMP parallel processing, you must specify the `-qsmp` compiler option.

- To choose automated parallelism, specify `-qsmp` or `-qsmp=auto`. This suboption enables the compiler to perform *implicit parallelism*, in addition to recognizing and implementing any OpenMP directives, library functions, and environment variables included in the program.
- To choose strict compliance to the OpenMP Specification 2.0, specify `-qsmp=omp`. This suboption ensures that the compiler implements only the OpenMP directives, library functions, and environment variables specified in the code. It does not perform any additional automated parallel processing.

Related References

- <http://www.openmp.org>
- "Pragmas to control parallel processing" in *XL C/C++ Compiler Reference*
- "Program parallelization" in *XL C/C++ Compiler Reference*

OpenMP directives


Each directive starts with `#pragma omp`, to reduce the potential for conflict with other pragma directives.

OpenMP directives in XL C/C++

Directive name	Directive Description
<code>parallel</code>	The <code>parallel</code> directive defines a <i>parallel region</i> , which is a region of the program that is to be executed by multiple threads in parallel.

OpenMP directives in XL C/C++

Directive name	Directive Description
for	The <code>for</code> directive identifies an iterative work-sharing construct that specifies a region in which the iterations of the associated loop should be executed in parallel. The iterations of the <code>for</code> loop are distributed across threads that already exist.
sections	The <code>sections</code> directive identifies a non-iterative work-sharing construct that specifies a set of constructs that are to be divided among threads in a team. Each section is executed once by a thread in the team.
single	The <code>single</code> directive identifies a construct that specifies that the associated structured block is executed by only one thread in the team (not necessarily the master thread).
parallel for	The <code>parallel for</code> directive is a shortcut form for a parallel region that contains a single <code>for</code> directive. The semantics are identical to explicitly specifying a <code>parallel</code> directive immediately followed by a <code>for</code> directive.
parallel sections	The <code>parallel sections</code> directive provides a shortcut form for specifying a parallel region containing a single <code>sections</code> directive. The semantics are identical to explicitly specifying a <code>parallel</code> directive immediately followed by a <code>sections</code> directive.
master	The <code>master</code> directive identifies a construct that specifies a structured block that is executed by the master thread of the team.
critical	The <code>critical</code> directive identifies a construct that restricts execution of the associated structured block to a single thread at a time. An optional <i>name</i> may be used to identify the critical region. A thread waits at the beginning of a critical region until no other thread is executing a critical region with the same name. All unnamed critical directives map to the same unspecified name.
barrier	The <code>barrier</code> directive synchronizes all the threads in a team. When encountered, each thread waits until all of the others have reached this point. After all threads have encountered the barrier, each thread begins executing the statements after the barrier directive in parallel.
atomic	The <code>atomic</code> directive identifies a specific memory location that must be updated atomically and not be exposed to multiple, simultaneous writing threads.
flush	The <code>flush</code> directive identifies a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory.
ordered	The <code>ordered</code> directive identifies a structured block of code that must be executed in sequential order.
threadprivate	The <code>threadprivate</code> directive declares file-scope, namespace-scope, or static block-scope variables to be private to a thread.

 The C compiler recognizes the following additional directives used for program parallelism. They are not part of the OpenMP Specification 1.0 or 2.0 and are not recognized by the C++ compiler.

- #pragma ibm critical
- #pragma ibm independent_calls
- #pragma ibm independent_loop
- #pragma ibm iterations
- #pragma ibm parallel_loop
- #pragma ibm permutation
- #pragma ibm schedule
- #pragma ibm sequential_loop

OpenMP data scope attribute clauses

Clauses may be specified on the directives to control the scope attributes of variables for the duration of the parallel or work-sharing constructs.

OpenMP data scope attribute clauses in XL C/C++

Data Scope Attribute Clause Name	Data Scope Attribute Clause Description
private	The private clause declares the variables in the list to be private to each thread in a team.
firstprivate	The firstprivate clause provides a superset of the functionality provided by the private clause.
lastprivate	The lastprivate clause provides a superset of the functionality provided by the private clause.
copyprivate	The copyprivate clause provides an alternative to using a shared variable to broadcast a value to a team. The mechanism uses a private variable to broadcast a value from one team member to other members.
num_threads	The num_threads clause provides the ability to request a specific number of threads for a parallel construct.
shared	The shared clause shares variables that appear in the list among all the threads in a team. All threads within a team access the same storage area for shared variables.
reduction	The reduction clause performs a reduction on the scalar variables that appear in list, with a specified operator.
default	The default clause allows the user to affect the data scope attributes of variables.

OpenMP library functions

OpenMP runtime library functions are included in the header `<omp.h>`. They include *execution environment functions* that can be used to control and query the parallel execution environment, and *lock functions* that can be used to synchronize access to data.

OpenMP runtime library functions in XL C/C++

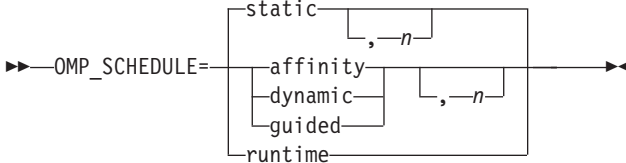
Runtime Library Function Name	Runtime Library Function Description
omp_set_num_threads	Sets the number of threads to use for subsequent parallel regions.
omp_get_num_threads	Returns the number of threads currently in the team executing the parallel region from which it is called.

OpenMP runtime library functions in XL C/C++

Runtime Library Function Name	Runtime Library Function Description
omp_get_max_threads	Returns the maximum value that can be returned by calls to omp_get_num_threads.
omp_get_thread_num	Returns the thread number, within its team, of the thread executing the function. The master thread of the team is thread 0.
omp_get_num_procs	Returns the maximum number of processors that could be assigned to the program.
omp_in_parallel	Returns non-zero if it is called within the dynamic extent of a parallel region executing in parallel; otherwise, it returns 0.
omp_set_dynamic	Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.
omp_get_dynamic	Returns non-zero if dynamic thread adjustment is enabled and returns 0 otherwise.
omp_set_nested	Enables or disables nested parallelism.
omp_get_nested	Returns non-zero if nested parallelism is enabled and 0 if it is disabled.
omp_init_lock	Initializes a simple lock.
omp_destroy_lock	Removes a simple lock.
omp_set_lock	Waits until a simple lock is available.
omp_unset_lock	Releases a simple lock.
omp_test_lock	Tests a simple lock.
omp_init_nest_lock	Initializes a nestable lock.
omp_destroy_nest_lock	Removes a nestable lock.
omp_set_nest_lock	Waits until a nestable lock is available.
omp_unset_nest_lock	Releases a nestable lock.
omp_test_nest_lock	Tests a nestable lock.
omp_get_wtick	Returns the number of seconds between successive clock ticks.
omp_get_wtime	Returns the elapsed wall-clock time in seconds.

OpenMP environment variables

OpenMP environment variables control the execution of parallel code. The names of environment variables must always be in upper case, while their values are not case-sensitive.

Description	Syntax
<p>OMP_SCHEDULE</p> <p>Sets the run-time schedule type and chunk size. Applies only to OpenMP directives that have the scheduling type set to runtime.</p>	<div style="text-align: center;">  </div> <p>where</p> <p>affinity An IBM extension valid for C only. Specifies that iterations of a loop are initially divided into local partitions of a size equal to the ceiling of the number of iterations divided by the number of threads: $\text{CEILING}(\text{number_of_iterations} \div \text{number_of_threads})$. Each local partition is further subdivided into chunks of a size equal to the ceiling of half of the number of iterations remaining in the local partition: $\text{CEILING}(\text{iterations_left_in_local_partition} \div 2)$. When a thread becomes free, it takes the next chunk from its local partition. If no chunks are in the local partition, the thread takes an available chunk from a partition of another thread. If n is specified, each local partition is subdivided into chunks of size n. If n is not specified, the default value is 1.</p> <p>dynamic Specifies that iterations for a for loop should be divided into a series of chunks of size n and that the chunks are handled according to the following process. A thread waiting for an assignment is assigned a chunk of iterations, which it executes and then waits for its next assignment. This process is repeated until all chunks are assigned. If n is not specified, the default chunk size is 1.</p> <p>guided Specifies that iterations for a for loop should be assigned to threads in chunks with decreasing sizes and that the chunks are handled according to the following process. A thread that finishes its assigned chunk of iterations is dynamically assigned another chunk, until all chunks are assigned. If n is not specified, the default value for the initial chunk size is 1.</p> <p>static Specifies that iterations for a for loop should be divided into a series of chunks of size n and that the chunks are handled according to the following process. Available threads are assigned chunks in an order determined by the thread number. When n is not specified, the iteration space is divided into chunks that are approximately equal in size, with one chunk assigned to each thread.</p> <p>n Is a positive number, representing the chunk size.</p>

Description	Syntax
<p>OMP_DYNAMIC</p> <p>Enables or disables dynamic adjustment of the number of threads available for the execution of parallel regions.</p>	<p>▶▶—OMP_DYNAMIC=<input type="checkbox"/>true <input type="checkbox"/>false▶▶</p> <p>where</p> <p>true Enables dynamic adjustment of the number of threads available.</p> <p>false Disables dynamic adjustment of the number of threads available.</p>
<p>OMP_NUM_THREADS</p> <p>Sets of the number of threads available for the execution.</p>	<p>▶▶—OMP_NUM_THREADS=<i>n</i>▶▶</p> <p>where</p> <p><i>n</i> Represents the number of threads.</p>
<p>OMP_NESTED</p> <p>Enables or disables nested parallelism.</p>	<p>▶▶—OMP_NESTED=<input type="checkbox"/>true <input type="checkbox"/>false▶▶</p> <p>where</p> <p>true Enables nested parallelism.</p> <p>false Disables nested parallelism.</p>

OpenMP implementation-defined behavior

The following information is not specified in the standard. Each implementation of the standard may have its own implementation-defined values.

Conditional Compilation

The `_OPENMP` macro is defined to 199810.

Scheduling

The `schedule` clause specifies how iterations of a `for` loop are divided among threads of the team. The possible OpenMP standard values are `static`, `dynamic`, `guided`, and `runtime`. In addition, IBM C adds the value `affinity` as an extension. In the absence of an explicitly defined `schedule` clause, the default schedule for XL C/C++ is `static`.

Tuning an OpenMP program

In addition to the environment variables that adjust various timings in the pthread life cycle, the following environment variables can be helpful for tuning an OpenMP application.

Selected environment variables for tuning OpenMP applications

Description	Syntax
<p>OMP_DYNAMIC</p> <p>Enables or disables dynamic adjustment of the number of threads available for the execution of parallel regions. When the variable is enabled (default setting), the run-time environment can adjust the number of threads it uses for executing parallel regions so it makes the most efficient use of system resources. The variable should be disabled for benchmarking, scaling tests, or if an application depends on a specific number of threads because dynamic checking can add a small amount of overhead.</p>	<p>►►—OMP_DYNAMIC=true false—►►</p> <p>where</p> <p>true Enables dynamic adjustment of the number of threads available.</p> <p>false Disables dynamic adjustment of the number of threads available.</p>
<p>MALLOCMULTIHEAP</p> <p>Multiple heaps are useful so that a threaded application can have more than one thread issuing memory allocation subroutine calls. With a single heap, all threads trying to do a <code>malloc()</code>, <code>free()</code>, or <code>realloc()</code> call would be serialized (that is, only one thread can do any of these three functions at a time). Such a situation could have a serious impact on multiprocessor machines. With multiple heaps, each thread gets its own heap, to a maximum of 32 separate heaps. Enable the use of multiple heaps by exporting this environment variable, which is not set by default.</p>	<p>►►—export—MALLOCMULTIHEAP—►►</p>
<p>XLSMPOPTS</p> <p>For 32-bit OpenMP applications, the default limit on stack size per thread is rather small and if it is exceeded it results in a run-time error. Should this occur, the stack size can be increased by setting the XLSMPOPTS environment variable with the stack suboption.</p>	<p>►►—export—XLSMPOPTS=stack=<i>n</i>—►►</p> <p>where</p> <p><i>n</i> Represents the stack size in bytes. The total stack size for all threads cannot exceed 256 MB (one memory segment). The one-segment limitation does not apply to 64-bit applications. Default value is 4 MB per thread.</p>

Related References

- “Tuning a multithreaded program” on page 56

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd. Laboratory
B3/KB7/8200/MKM
8200 Warden Avenue
Markham, Ontario, Canada L6G 1C7

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1998, 2004. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow the customer to write application software that obtains the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification, and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and service marks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX	POWER3	pSeries
@server	POWER4	Redbooks
IBM	POWER5	RS/6000
	PowerPC	VisualAge
	PowerPC Architecture	

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names, may be trademarks or service marks of others.

Industry standards

The following standards are supported:

- The C language is consistent with the International Standard C (ANSI/ISO-IEC 9899–1990 [1992]). This standard has officially replaced American National Standard for Information Systems-Programming Language C (X3.159–1989) and is technically equivalent to the ANSI C standard. The compiler supports the changes adopted into the C Standard by ISO/IEC 9899:1990/Amendment 1:1994.
- The C language is consistent with the International Standard for Information Systems-Programming Language C (ISO/IEC 9899–1999 (E)).
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998), the first formal definition of the language.
- The C++ language is also consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:2002 (E)), currently referred to as *Standard C++*.
- The C and C++ compilers support the OpenMP C and C++ Application Programming Interface Version 2.0.