

XL C/C++ Enterprise Edition for AIX



# Programming Guide

*Version 7.0*



XL C/C++ Enterprise Edition for AIX



# Programming Guide

*Version 7.0*

**Note!**

Before using this information and the product it supports, read the information in "Notices" on page 95.

**First Edition (July 2004)**

This edition applies to version 7.0 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 to [compinfo@ca.ibm.com](mailto:compinfo@ca.ibm.com). Be sure to include your e-mail address if you want a reply. Include the title and order number of this book, and the page number or topic related to your comment.

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 1998, 2004. All rights reserved.

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

---

# Contents

|   |          |
|---|----------|
| <b>About this guide</b> . . . . .         | <b>v</b> |
| Document conventions . . . . .            | v        |
| Highlighting conventions . . . . .        | v        |
| Icons . . . . .                           | vi       |
| How to read the syntax diagrams . . . . . | vi       |

## **Chapter 1. Using 32-bit and 64-bit modes** . . . . . **1**

|   |   |
|---|---|
| Assigning long values . . . . .                       | 2 |
| Assigning constant values to long variables . . . . . | 2 |
| Bit-shifting long values . . . . .                    | 3 |
| Assigning pointers . . . . .                          | 3 |
| Aligning aggregate data . . . . .                     | 4 |
| Calling Fortran code . . . . .                        | 4 |

## **Chapter 2. Aligning data in aggregates** . . . . . **5**

|   |    |
|---|----|
| Using alignment modes and modifiers . . . . . | 5  |
| General rules for alignment . . . . .         | 7  |
| Alignment examples . . . . .                  | 7  |
| Using and aligning bit fields . . . . .       | 8  |
| Rules for natural alignment . . . . .         | 9  |
| Rules for power alignment . . . . .           | 9  |
| Rules for Mac68K alignment . . . . .          | 9  |
| Rules for bit-packed alignment . . . . .      | 10 |
| Example of bit field alignment . . . . .      | 10 |

## **Chapter 3. Handling floating point operations** . . . . . **11**

|  |    |
|--|----|
| Handling multiply-add operations . . . . .                     | 11 |
| Handling floating-point rounding . . . . .                     | 11 |
| Handling floating-point exceptions . . . . .                   | 12 |
| Single-precision and double-precision performance . . . . .    | 13 |
| Using the Mathematical Acceleration Subsystem (MASS) . . . . . | 13 |
| Using the scalar library . . . . .                             | 13 |
| Using the vector libraries . . . . .                           | 14 |
| Compiling and linking a program with MASS . . . . .            | 17 |

## **Chapter 4. Using memory heaps.** . . . . **19**

|   |    |
|---|----|
| Managing memory with multiple heaps . . . . .                     | 19 |
| Functions for managing user-created heaps . . . . .               | 20 |
| Creating a heap . . . . .   | 21 |
| Expanding a heap . . . . .  | 22 |
| Using a heap . . . . .  | 23 |
| Getting information about a heap . . . . .                        | 24 |
| Closing and destroying a heap . . . . .                           | 24 |
| Changing the default heap used in a program . . . . .             | 25 |
| Compiling and linking a program with user-created heaps . . . . . | 25 |
| Examples of creating and using user heaps . . . . .               | 25 |
| Debugging memory heaps . . . . .                                  | 30 |
| Functions for checking memory heaps . . . . .                     | 31 |
| Functions for debugging memory heaps . . . . .                    | 31 |
| Using memory allocation fill patterns . . . . .                   | 33 |

|                                  |    |
|----------------------------------|----|
| Skipping heap checking . . . . . | 33 |
| Using stack traces . . . . .     | 34 |

## **Chapter 5. Using C++ templates** . . . . . **35**

|  |    |
|--|----|
| Using the -qtempinc compiler option . . . . .            | 35 |
| Example of -qtempinc . . . . .                           | 36 |
| Regenerating the template instantiation file . . . . .   | 38 |
| Using -qtempinc with shared libraries . . . . .          | 38 |
| Using the -qtemplateregistry compiler option . . . . .   | 38 |
| Recompiling related compilation units . . . . .          | 38 |
| Switching from -qtempinc to -qtemplateregistry . . . . . | 39 |

## **Chapter 6. Ensuring thread safety (C++)** **41**

|  |    |
|--|----|
| Ensuring thread safety of template objects . . . . . | 41 |
| Ensuring thread safety of stream objects . . . . .   | 41 |

## **Chapter 7. Constructing a library** . . . . . **43**

|   |    |
|---|----|
| Compiling and linking a library . . . . .   | 43 |
| Compiling a static library . . . . .  | 43 |
| Compiling a shared library . . . . .  | 43 |
| Linking a shared library to another shared library . . . . .                      | 45 |
| Initializing static objects in libraries (C++) . . . . .                          | 45 |
| Assigning priorities to objects . . . . .   | 46 |
| Order of object initialization across libraries . . . . .                         | 48 |
| Dynamically loading a shared library . . . . .                                    | 49 |
| Loading and initializing a module with the loadAndInit function . . . . .         | 50 |
| Terminating and unloading a module with the terminateAndUnload function . . . . . | 51 |

## **Chapter 8. Using the C++ utilities** . . . . . **53**

|   |    |
|---|----|
| Demangling compiled C++ names . . . . .                                 | 53 |
| Demangling compiled C++ names with c++filt . . . . .                    | 53 |
| Demangling compiled C++ names with the demangle class library . . . . . | 54 |
| Creating a shared library with the makeC++SharedLib utility . . . . .   | 56 |
| Linking with the linkx1C utility . . . . .                              | 57 |

## **Chapter 9. Optimizing your applications** **59**

|   |    |
|---|----|
| Using optimization levels . . . . .                           | 60 |
| Getting the most out of optimization levels 2 and 3 . . . . . | 62 |
| Optimizing for system architecture . . . . .                  | 63 |
| Getting the most out of target machine options . . . . .      | 63 |
| Using high-order loop analysis and transformations . . . . .  | 64 |
| Getting the most out of -qhot . . . . .                       | 65 |
| Using shared-memory parallelism . . . . .                     | 65 |
| Getting the most out of -qsm . . . . .                        | 65 |
| Using interprocedural analysis . . . . .                      | 66 |
| Getting the most from -qipa . . . . .                         | 67 |
| Using profile-directed feedback . . . . .                     | 67 |
| Example of compilation with pdf and showpdf . . . . .         | 69 |
| Other optimization options . . . . .                          | 70 |

|   |    |
|---|----|
| Summary of options for optimization and performance . . . . . | 71 |
|---|----|

**Chapter 10. Coding your application to improve performance . . . . . 73**

|  |    |
|--|----|
| Find faster input/output techniques . . . . .    | 73 |
| Reduce function-call overhead . . . . .          | 73 |
| Manage memory efficiently . . . . .              | 75 |
| Optimize variables . . . . .                     | 75 |
| Manipulate strings efficiently . . . . .         | 76 |
| Optimize expressions and program logic . . . . . | 77 |
| Optimize operations in 64-bit mode . . . . .     | 77 |

**Appendix. Memory debug library functions. . . . . 79**

|   |    |
|---|----|
| Memory allocation debug functions . . . . .                                       | 79 |
| _debug_calloc — Allocate and initialize memory . . . . .                          | 79 |
| _debug_free — Free allocated memory . . . . .                                     | 80 |
| _debug_heapmin — Free unused memory in the default heap . . . . .                 | 81 |
| _debug_malloc — Allocate memory . . . . .   | 82 |
| _debug_ucalloc — Reserve and initialize memory from a user-created heap . . . . . | 83 |

|  |    |
|--|----|
| _debug_uheapmin — Free unused memory in a user-created heap . . . . .    | 84 |
| _debug_umalloc — Reserve memory blocks from a user-created heap. . . . . | 84 |
| _debug_realloc — Reallocate memory block . . . . .                       | 85 |
| String handling debug functions . . . . .                                | 87 |
| _debug_memcpy — Copy bytes . . . . .                                     | 87 |
| _debug_memmove — Copy bytes . . . . .                                    | 88 |
| _debug_memset — Set bytes to value. . . . .                              | 89 |
| _debug_strcat — Concatenate strings. . . . .                             | 89 |
| _debug_strcpy — Copy strings . . . . .                                   | 90 |
| _debug_strncat — Concatenate strings . . . . .                           | 91 |
| _debug_strncpy — Copy strings . . . . .                                  | 92 |
| _debug_strnset — Set characters in a string. . . . .                     | 93 |
| _debug_strset — Set characters in a string . . . . .                     | 94 |

**Notices . . . . . 95**

|   |    |
|---|----|
| Programming interface information . . . . . | 96 |
| Trademarks and service marks . . . . .      | 97 |
| Industry standards . . . . .                | 97 |

---

## About this guide

This guide discusses advanced topics related to the use of the IBM® XL C/C++ Enterprise Edition for AIX® compiler, with a particular focus on program portability and optimization. The guide provides both reference information and practical tips for getting the most out of the compiler's capabilities, through recommended programming practices and compilation procedures. The guide also contains extensive cross-references to the relevant sections of the other reference guides in the XL C/C++ Enterprise Edition for AIX documentation set.

This guide includes these topics:

- Chapter 1, "Using 32-bit and 64-bit modes," on page 1 discusses common problems that arise when porting existing 32-bit applications to 64-bit mode, and provides recommendations for avoiding these problems.
- Chapter 2, "Aligning data in aggregates," on page 5 discusses the different compiler options available for controlling the alignment of data in aggregates, such as structures and classes, on all platforms.
- Chapter 3, "Handling floating point operations," on page 11 discusses options available for controlling the way floating-point operations are handled by the compiler.
- Chapter 4, "Using memory heaps," on page 19 discusses compiler library functions for heap memory management, including using custom memory heaps, and validating and debugging heap memory.
- Chapter 5, "Using C++ templates," on page 35 discusses the different options for compiling programs that include C++ templates.
- Chapter 6, "Ensuring thread safety (C++)," on page 41 discusses thread-safety issues related to C++ class libraries, including input/output streams, and standard templates.
- Chapter 7, "Constructing a library," on page 43 discusses how to compile and link static and shared libraries, and how to specify the initialization order of static objects in C++ programs.
- Chapter 8, "Using the C++ utilities," on page 53 discusses some additional utilities shipped with XL C/C++ Enterprise Edition for AIX, for demangling compiled symbol names, creating shared libraries, and linking C++ modules.
- Chapter 9, "Optimizing your applications," on page 59 discusses the various options provided by the compiler for optimizing your programs, and provides recommendations for use of the different options.
- Chapter 10, "Coding your application to improve performance," on page 73 discusses recommended programming practices and coding techniques for enhancing program performance and compatibility with the compiler's optimization capabilities.
- "Memory debug library functions," on page 79 provides a reference listing and examples of all compiler debug memory library functions.

---

## Document conventions

### Highlighting conventions

This guide uses the following highlighting conventions:

|                  |  |
|------------------|--|
| <b>Bold</b>      | Identifies commands, keywords, file, directory, and path names, environment variables, executable names, 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.            |
| <b>Monospace</b> | Identifies examples of program code.   |

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 language constructs. Some examples are only code fragments and will not compile without additional code.

## Icons

In general, this guide documents XL C/C++ functionality as it has been implemented on the AIX platform. However, where issues are discussed that affect portability to other platforms, the following icons are used:



Indicates the functionality supported on the AIX platform.



Indicates the functionality supported on the Linux<sup>®</sup> platform.



Indicates the functionality supported on the Mac OS X platform.



Indicates a feature that is supported only in the C++ language.



Indicates a feature that is supported only in the C language.

---

## 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.
  - The ►— symbol indicates that a command, directive, or statement is continued from the previous line.
  - The —►◄ symbol indicates the end of a command, directive, or statement.
- Diagrams of syntactical units other than complete commands, directives, or statements start with the ►— symbol and end with the —► 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.

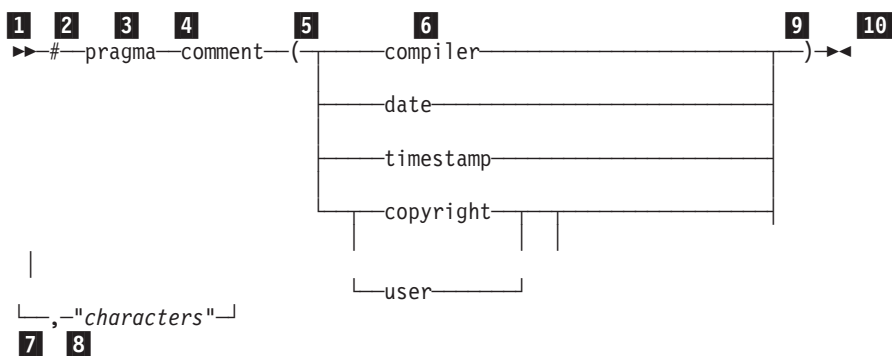
- Keywords appear in nonitalic letters and should be entered exactly as shown (for example, `extern`).

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.

The following syntax diagram example shows the syntax for the **#pragma comment** directive.

## Reading the Syntax Diagrams



- 1** This is the start of the syntax diagram.
- 2** The symbol `#` must appear first.
- 3** The keyword `pragma` must appear following the `#` symbol.
- 4** The name of the pragma comment must appear following the keyword `pragma`.
- 5** An opening parenthesis must be present.
- 6** The comment type must be entered only as one of the types indicated: `compiler`, `date`, `timestamp`, `copyright`, or `user`.
- 7** A comma must appear between the comment type `copyright` or `user`, and an optional character string.
- 8** A character string must follow the comma. The character string must be enclosed in double quotation marks.
- 9** A closing parenthesis is required.
- 10** This is the end of the syntax diagram.

The following examples of the **#pragma comment** directive are syntactically correct according to the diagram shown above:

```
#pragma  
comment(date)  
#pragma comment(user)  
#pragma comment(copyright,"This text will appear in the module")
```

---

## Chapter 1. Using 32-bit and 64-bit modes

You can use XL C/C++ to develop both 32-bit and 64-bit applications. To do so, specify **-q32** (the default) or **-q64**, respectively, during compilation. Alternatively, you can set the **OBJECT\_MODE** environment variable to 32 or 64.

However, porting existing applications from 32-bit to 64-bit mode can lead to a number of problems, mostly related to the differences in C/C++ long and pointer data type sizes and alignment between the two modes. The following table summarizes these differences.

*Table 1. Size and alignment of data types in 32-bit and 64-bit modes*

| Data type                             | 32-bit mode |                   | 64-bit mode |                   |
|---------------------------------------|-------------|-------------------|-------------|-------------------|
|                                       | Size        | Alignment         | Size        | Alignment         |
| long, unsigned long                   | 4 bytes     | 4-byte boundaries | 8 bytes     | 8-byte boundaries |
| pointer                               | 4 bytes     | 4-byte boundaries | 8 bytes     | 8-byte boundaries |
| size_t (system-defined unsigned long) | 4 bytes     | 4-byte boundaries | 8 bytes     | 8-byte boundaries |
| ptrdiff_t (system-defined long)       | 4 bytes     | 4-byte boundaries | 8 bytes     | 8-byte boundaries |

The following sections discuss some of the common pitfalls implied by these differences, as well as recommended programming practices to help you avoid most of these issues:

- “Assigning long values” on page 2
- “Assigning pointers” on page 3
- “Aligning aggregate data” on page 4
- “Calling Fortran code” on page 4

When compiling in 32-bit or 64-bit mode, you can use the **-qwarn64** option to help diagnose some issues related to porting applications. In either mode, the compiler immediately issues a warning if undesirable results, such as truncation or data loss, have occurred.

For suggestions on improving performance in 64-bit mode, see “Optimize operations in 64-bit mode” on page 77.

---

### Related references

- **-q32/-q64** in *XL C/C++ Compiler Reference*
- **-qwarn64** in *XL C/C++ Compiler Reference*
- “Set Environment Variables to Select 64- or 32-bit Modes” in *XL C/C++ Compiler Reference*

## Assigning long values

The limits of **long** type integers defined in the **limits.h** standard library header file are different in 32-bit and 64-bit modes, as shown in the following table.

Table 2. Constant limits of long integers in 32-bit and 64-bit modes

| Symbolic constant                    | Mode   | Value       | Hexadecimal          | Decimal                     |
|--------------------------------------|--------|-------------|----------------------|-----------------------------|
| LONG_MIN<br>(smallest signed long)   | 32-bit | $-(2^{31})$ | 0x80000000L          | -2,147,483,648              |
|                                      | 64-bit | $-(2^{63})$ | 0x8000000000000000L  | -9,223,372,036,854,775,808  |
| LONG_MAX<br>(longest signed long)    | 32-bit | $2^{31}-1$  | 0x7FFFFFFFL          | +2,147,483,647              |
|                                      | 64-bit | $2^{63}-1$  | 0x7FFFFFFFFFFFFFFFL  | +9,223,372,036,854,775,807  |
| ULONG_MAX<br>(longest unsigned long) | 32-bit | $2^{32}-1$  | 0xFFFFFFFFUL         | +4,294,967,295              |
|                                      | 64-bit | $2^{64}-1$  | 0xFFFFFFFFFFFFFFFFUL | +18,446,744,073,709,551,615 |

Implications of these differences are:

- Assigning a long value to a **double** variable can cause loss of accuracy.
- Assigning constant values to long-type variables can lead to unexpected results. This issue is explored in more detail in “Assigning constant values to long variables.”
- Bit-shifting long values will produce different results, as described in “Bit-shifting long values” on page 3.
- Using **int** and **long** types interchangeably in expressions will lead to implicit conversion through promotions, demotions, assignments, and argument passing, and can result in truncation of significant digits, sign shifting, or unexpected results, without warning.

In situations where a long-type value can overflow when assigned to other variables or passed to functions, you must:

- Avoid implicit type conversion by using explicit type casting to change types.
- Ensure that all functions that return long types are properly prototyped.
- Ensure that long parameters can be accepted by the functions to which they are being passed.

## Assigning constant values to long variables

Although type identification of constants follows explicit rules in C and C++, many programs use hexadecimal or unsuffixed constants as “typeless” variables and rely on a two’s complement representation to exceed the limits permitted on a 32-bit system. As these large values are likely to be extended into a 64-bit **long** type in 64-bit mode, unexpected results can occur, generally at boundary areas such as:

- constant  $\geq$  **UINT\_MAX**
- constant  $<$  **INT\_MIN**
- constant  $>$  **INT\_MAX**

Some examples of unexpected boundary side effects are listed in the following table.

Table 3. Unexpected boundary results of constants assigned to long types

| Constant assigned to long | Equivalent value | 32 bit mode    | 64 bit mode    |
|---------------------------|------------------|----------------|----------------|
| -2,147,483,649            | INT_MIN-1        | +2,147,483,647 | -2,147,483,649 |
| +2,147,483,648            | INT_MAX+1        | -2,147,483,648 | +2,147,483,648 |
| +4,294,967,726            | UINT_MAX+1       | 0              | +4,294,967,296 |
| 0xFFFFFFFF                | UINT_MAX         | -1             | +4,294,967,295 |
| 0x100000000               | UINT_MAX+1       | 0              | +4,294,967,296 |
| 0xFFFFFFFFFFFFFFFF        | ULONG_MAX        | -1             | -1             |

Unsuffixes constants can lead to type ambiguities that can affect other parts of your program, such as when the results of `sizeof` operations are assigned to variables. For example, in 32-bit mode, the compiler types a number like 4294967295 (`UINT_MAX`) as an unsigned long and `sizeof` returns 4 bytes. In 64-bit mode, this same number becomes a signed long and `sizeof` will return 8 bytes. Similar problems occur when passing constants directly to functions.

You can avoid these problems by using the suffixes `L` (for long constants) or `UL` (for unsigned long constants) to explicitly type all constants that have the potential of affecting assignment or expression evaluation in other parts of your program. In the example cited above, suffixing the number as 4294967295U forces the compiler to always recognize the constant as an **unsigned int** in 32-bit or 64-bit mode.

## Bit-shifting long values

Left bit-shifting long values will produce different results in 32-bit and 64-bit modes. The examples in the table below show the effects of performing a bit-shift on long constants, using the following code segment:

```
long l=valueL<<1;
```

Table 4. Results of bit-shifting long values

| Initial value | Symbolic constant | Value after bit shift |                    |
|---------------|-------------------|-----------------------|--------------------|
|               |                   | 32-bit mode           | 64-bit mode        |
| 0x7FFFFFFFL   | INT_MAX           | 0xFFFFFFFFE           | 0x00000000FFFFFFFE |
| 0x80000000L   | INT_MIN           | 0x00000000            | 0x0000000100000000 |
| 0xFFFFFFFFL   | UINT_MAX          | 0xFFFFFFFFE           | 0x1FFFFFFFE        |

---

## Assigning pointers

In 64-bit mode, pointers and `int` types are no longer the same size. The implications of this are:

- Exchanging pointers and `int` types causes segmentation faults.
- Passing pointers to a function expecting an `int` type results in truncation.
- Functions that return a pointer, but are not explicitly prototyped as such, return an `int` instead and truncate the resulting pointer, as illustrated in the following example.

Although code constructs such as the following are valid in 32-bit mode:

```
a=(char*) calloc(25);
```

without a function prototype for `calloc`, the compiler assumes the function returns an `int`, so `a` is silently truncated, and then sign-extended. Type casting the result will not prevent the truncation, as the address of the memory allocated by `calloc` was already truncated during the return. In this example, the correct solution would be to include the appropriate header file, `stdlib.h`, which contains the prototype for `calloc`.

To avoid these types of problems:

- Prototype any functions that return a pointer.
- Be sure that the type of parameter you are passing in a function (pointer or `int`) call matches the type expected by the function being called.
- For applications that treat pointers as an integer type, use type `long` or `unsigned long` in either 32-bit or 64-bit mode.

## Aligning aggregate data

Structures are aligned according to the strictest aligned member in both 32-bit and 64-bit modes. However, since long types and pointers change size and alignment in 64-bit, the alignment of a structure's strictest member can change, resulting in changes to the alignment of the structure itself.

Structures that contain pointers or long types cannot be shared between 32-bit and 64-bit applications. Unions that attempt to share `long` and `int` types, or overlay pointers onto `int` types can change or corrupt the alignment. In general, you should check all but the simplest structures for alignment and size dependencies.

In 64-bit mode, member values in a structure passed by value to a `va_arg` argument might not be accessed properly if the size of the structure is not a multiple of 8-bytes. This is a known limitation of the operating system.

For detailed information on aligning data structures, including structures that contain bit fields, see Chapter 2, "Aligning data in aggregates," on page 5.

## Calling Fortran code

A significant number of applications use C, C++, and Fortran together, by calling each other or sharing files. It is currently easier to modify data sizes and types on the C side than the on Fortran side of such applications. The following table lists C and C++ types and the equivalent Fortran types in the different modes.

*Table 5. Equivalent C/C++ and Fortran data types*

| C/C++ type    | Fortran type |                     |
|---------------|--------------|---------------------|
|               | 32-bit       | 64-bit              |
| signed int    | INTEGER      | INTEGER             |
| signed long   | INTEGER      | INTEGER*8           |
| unsigned long | LOGICAL      | LOGICAL*8           |
| pointer       | INTEGER      | INTEGER*8           |
|               |              | POINTER (4 bytes)   |
|               |              | POINTER*8 (8 bytes) |

---

## Chapter 2. Aligning data in aggregates

XL C/C++ provides many mechanisms for specifying data alignment at the levels of individual variables, members of aggregates, entire aggregates, and entire compilation units. If you are porting applications between different platforms, or between 32-bit and 64-bit modes, you will need to take into account the differences between alignment settings available in the different environments, to prevent possible data corruption and deterioration in performance.

“Using alignment modes and modifiers” discusses the default alignment settings for all data types on the different platforms and addressing models; options you can use to control the alignment of aggregates and aggregate members; and general rules for aggregate alignment. This section also provides examples of structure layouts based on the different alignment options.

“Using and aligning bit fields” on page 8 discusses additional rules and considerations for the use and alignment of bit fields, and provides an example of bit-packed alignment.

---

### Using alignment modes and modifiers

Within aggregates that contain different data types, including C and C++ structures and unions, and C++ classes, each data type supported by XL C/C++ is aligned along byte boundaries according to platform-specific defaults, as follows:

- ▶ **AIX** **power** or **full**, which are equivalent.
- ▶ **Linux** **linuxppc**.
- ▶ **Mac OS X** **power**.

Each of these settings is defined in Table 6 on page 6.

You can also explicitly control the alignment of data by using an alignment *mode*, as well as alignment *modifiers*. Alignment *modes* allow you to do the following:

#### Set the alignment for all aggregates in a single file or multiple files in the compilation process

To use this approach, you specify the **-qalign** compiler option during compilation. The valid suboptions for **-qalign** for each platform are provided in Table 6 on page 6.

#### Set the alignment for a single aggregate or multiple aggregates in a file

To use this approach, you specify the **#pragma align** or **#pragma options align** directives in the source files. The valid suboptions for **#pragma align** for each platform are provided in Table 6 on page 6. Each directive changes the alignment rule in effect for all aggregates that follow the directive until another directive is encountered, or until the end of the compilation unit.

#### Set the alignment for a single aggregate

In addition to the **#pragma align** directive, you can use the following in source files:

- Include the **\_\_attribute\_\_((aligned(n)))** type attribute in structure declarations. The value for *n* must be a positive power of 2. For the

correct syntax for using `__attribute__((aligned))` as a type attribute for an aggregate, see "Type Attributes" in *XL C/C++ Language Reference*.

- Include the `__align(n)` specifier in structure declarations. The value for *n* must be a positive power of 2.

Alignment *modifiers* allow you to do the following:

#### Set the alignment for all members in an aggregate

To use this approach, you can use any of the following in source files:

- Include the `#pragma pack` directive before structure declarations. For valid values for this directive, see `#pragma pack` in *XL C/C++ Compiler Reference*.
- Include the `__attribute__((packed))` type attribute in structure declarations. For the correct syntax for using `__attribute__((packed))` as a type attribute, see "Type Attributes" in *XL C/C++ Language Reference*.

#### Set the alignment for a single member within an aggregate

To use this approach, include `__attribute__((packed))` or `__attribute__((aligned(n)))` type or variable attributes in structure declarations. The value for *n* in `__attribute__((aligned(n)))` must be a positive power of 2. For more information on the variable attributes, see "The aligned Variable Attribute" and "The packed Variable Attribute" in *XL C/C++ Language Reference*. For information on the type attributes, see "Type Attributes" in *XL C/C++ Language Reference*.

Table 6. Alignment settings

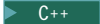
| Data type   | Storage  | Alignment settings and supported platforms |                       |      |                            |         |          |                         |                     |
|---|----------|--|-----------------------|------|----------------------------|---------|----------|-------------------------|---------------------|
|   |          | natural                                    | power                 | full | mac68k                     | twobyte | linuxppc | bit_packed <sup>3</sup> | packed <sup>3</sup> |
|   |          | AIX Mac                                    | AIX Mac               | AIX  | AIX Mac                    | AIX     | Linux    | AIX Mac Linux           | AIX                 |
| <code>_Bool</code> (C), <code>bool</code> (C++)                           | 1 byte   | 1 byte                                     | 1 byte                |      | 1 byte                     |         | n/a      | 1 byte                  |                     |
| <code>char</code> , signed <code>char</code> , unsigned <code>char</code> | 1 byte   | 1 byte                                     | 1 byte                |      | 1 byte                     |         | 1 byte   | 1 byte                  |                     |
| <code>wchar_t</code> (32-bit mode)  | 2 bytes  | 2 bytes                                    | 2 bytes               |      | 2 bytes                    |         | 2 bytes  | 1 byte                  |                     |
| <code>wchar_t</code> (64-bit mode)  | 4 bytes  | 4 bytes                                    | 4 bytes               |      | not supported <sup>2</sup> |         | 4 bytes  | 1 byte                  |                     |
| <code>int</code> , unsigned <code>int</code>                              | 4 bytes  | 4 bytes                                    | 4 bytes               |      | 2 bytes                    |         | 4 bytes  | 1 byte                  |                     |
| short <code>int</code> , unsigned short <code>int</code>                  | 2 bytes  | 2 bytes                                    | 2 bytes               |      | 2 bytes                    |         | 2 bytes  | 1 byte                  |                     |
| long <code>int</code> , unsigned long <code>int</code> (32-bit mode)      | 4 bytes  | 4 bytes                                    | 4 bytes               |      | 2 bytes                    |         | 4 bytes  | 1 byte                  |                     |
| long <code>int</code> , unsigned long <code>int</code> (64-bit mode)      | 8 bytes  | 8 bytes                                    | 8 bytes               |      | not supported <sup>2</sup> |         | 8 bytes  | 1 byte                  |                     |
| long long   | 8 bytes  | 8 bytes                                    | 8 bytes               |      | 2 bytes                    |         | 8 bytes  | 1 byte                  |                     |
| <code>float</code>  | 4 bytes  | 4 bytes                                    | 4 bytes               |      | 2 bytes                    |         | 4 bytes  | 1 byte                  |                     |
| <code>double</code>   | 8 bytes  | 8 bytes                                    | see note <sup>1</sup> |      | 2 bytes                    |         | 8 bytes  | 1 byte                  |                     |
| long double   | 8 bytes  | 8 bytes                                    | see note <sup>1</sup> |      | 2 bytes                    |         | 8 bytes  | 1 byte                  |                     |
| long double with <code>-qlongdouble</code>                                | 16 bytes | 16 bytes                                   | see note <sup>1</sup> |      | 2 bytes                    |         | n/a      | 1 byte                  |                     |
| pointer (32-bit mode)   | 4 bytes  | 4 bytes                                    | 4 bytes               |      | 2 bytes                    |         | 4 bytes  | 1 byte                  |                     |
| pointer (64-bit mode)   | 8 bytes  | 8 bytes                                    | 8 bytes               |      | not supported <sup>2</sup> |         | 8 bytes  | 1 byte                  |                     |



Table 6. Alignment settings (continued)

| Data type  | Storage | Alignment settings and supported platforms |            |      |            |         |          |                         |                     |
|--|---------|--|------------|------|------------|---------|----------|-------------------------|---------------------|
|  |         | natural                                    | power      | full | mac68k     | twobyte | linuxppc | bit_packed <sup>3</sup> | packed <sup>3</sup> |
|  |         | AIX<br>Mac                                 | AIX<br>Mac | AIX  | AIX<br>Mac | AIX     | Linux    | AIX<br>Mac<br>Linux     | AIX                 |
| <b>Notes:</b> <ol style="list-style-type: none"> <li>1. These types use the <b>natural</b> alignment for the first member in the aggregate and 4 bytes or the <b>natural</b> alignment (whichever is less) for subsequent members.</li> <li>2. If you declare an aggregate with a member of this type and try to compile with this alignment setting, the compiler issues a warning message, and compiles with the default alignment setting for the appropriate platform.</li> <li>3. The <b>packed</b> alignment will not pack bit-field members at the bit level; use the <b>bit_packed</b> alignment if you want to pack bit fields at the bit level.</li> </ol> |         |  |            |      |            |         |          |                         |                     |

If you generate data with an application on one platform and read the data with an application on another platform, you will want to be sure to use a platform-neutral alignment mode, such as `#pragma pack` or `qalign=bit_packed`.

**Note:**  The C++ compiler might generate extra fields for classes that contain base classes or virtual functions. Objects of these types might not conform to the usual mappings for aggregates.

## General rules for alignment

If you control the alignment of aggregates with any of the settings listed in Table 6 on page 6, the following rules apply:

- For all alignment settings, the *size* of an aggregate is the smallest multiple of its alignment value that can encompass all of the members of the aggregate.
- For all alignment settings except **mac68k**, the *alignment* of an aggregate is equal to the largest alignment value of any of its members.
- For **mac68k** alignment, any aggregate has an alignment of 2 bytes, regardless of the data types of its members.
- Aligned aggregates can be nested, and the alignment rules applicable to each nested aggregate are determined by the alignment mode that is in effect when a nested aggregate is declared.

For rules on aligning aggregates containing bit fields, see “Using and aligning bit fields” on page 8.

## Alignment examples

The following examples use these symbols to show padding and boundaries:

p = padding

| = halfword (2-byte) boundary

: = byte boundary

### Mac68K example

For:

```
#pragma options align=mac68k
struct B {
    char a;
    double b;
}
#pragma options align=reset
```

The size of B is 10 bytes. The alignment of B is 2 bytes. The layout of B is:

```
|a:p|b:b|b:b|b:b|b:b|
```

### Packed example

For:

```
#pragma options align=packed
struct {
    char a;
    double b;
} B;
#pragma options align=reset
```

The size of B is 9 bytes. The layout of B is:

```
|a:b|b:b|b:b|b:
```

### Nested aggregate example

For:

```
#pragma options align=mac68k
struct A {
    char a;
    #pragma options align=power
    struct B {
        int b;
        char c;
    } B1; // <-- B1 laid out using Power alignment rules
    #pragma options align=reset // <-- has no effect on A or B,
                                // but on subsequent structs
    char d;
};
#pragma options align=reset
```

The size of A is 12 bytes. The alignment of A is 2 bytes. The layout of A is:

```
|a:p|b:b|b:b|c:p|p:p|d:p|
```

## Related references

- `-qalign` in *XL C/C++ Compiler Reference*
- `#pragma align` in *XL C/C++ Compiler Reference*
- `#pragma pack` in *XL C/C++ Compiler Reference*
- "The **aligned** Variable Attribute", "The **packed** Variable Attribute", "The **\_\_align** Specifier", and "Type Attributes" in "Declarations" in *XL C/C++ Language Reference*.

---

## Using and aligning bit fields

You can declare a bit field as a `_Bool` (C), `bool` (C++), `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, or `unsigned long long` data type. A bit field is always 4 or 8 bytes, depending on the declared base type and the compilation mode (32-bit or 64-bit).



**C** In the C language, you can specify bit fields as **char** or **short** instead of **int**, but XL C/C++ maps them as if they were **unsigned int**. The length of a bit field cannot exceed the length of its base type. In extended mode, you can use the **sizeof** operator on a bit field. (The **sizeof** operator on a bit field always returns 4.)

**C++** The length of a bit field can exceed the length of its base type, but the remaining bits will be used to pad the field, and will not actually store any value.

However, alignment rules for aggregates containing bit fields are different depending on the alignment setting you specify. These rules are described below.

## Rules for natural alignment

- A zero-length bit field pads to the next alignment boundary of its base declared type. This causes the next member to begin on a 4-byte boundary for all types except **long** in 64-bit mode and **long long** in both 32-bit and 64-bit mode, which will move the next member to the next 8-byte boundary. Padding does not occur if the previous member's memory layout ended on the appropriate boundary.
- **C** An aggregate that contains only zero-length bit fields has a length of 0 bytes and an alignment of 4 bytes.
- **C++** An aggregate that contains only zero-length bit fields has a length of 4 or 8 bytes, depending on the declared type of the bit field and the compilation mode.

## Rules for power alignment

- Aggregates containing bit fields are 4-byte (word) aligned.
- Bit fields are packed into the current word. If a bit field would cross a word boundary, it starts at the next word boundary.
- A bit field of length zero causes the bit field that immediately follows it to be aligned at the next word boundary, or 8 bytes, depending on the declared type and the compilation mode. If the zero-length bit field is at a word boundary, the next bit field starts at this boundary.
- **C** An aggregate that contains only zero-length bit fields has a length of 0 bytes.
- **C++** An aggregate that contains only zero-length bit fields has the length of 1 byte.

## Rules for Mac68K alignment

- Bit fields are packed into a word and are aligned on a 2-byte boundary.
- Bit fields that would cross a word boundary are moved to the *next* halfword boundary even if they are already starting on a halfword boundary. (The bit field can still end up crossing a word boundary.)
- A bit field of length zero forces the next member (even if it is not a bit field) to start at the *next* halfword boundary even if the zero-length bit field is currently at a halfword boundary.
- An aggregate containing nothing but zero-length bit fields has a length, in bytes, of two times the number of zero-length bit fields.
- For unions, there is one special case: unions whose largest element is a bit field of length 16 or less have a size of 2 bytes. If the length of the bit field is greater than 16, the size of the union is 4 bytes.

## Rules for bit-packed alignment

- Bit fields have an alignment of 1 byte, and are packed with no default padding between bit fields.
- A zero-length bit field causes the next member to start at the next byte boundary. If the zero-length bit field is already at a byte boundary, the next member starts at this boundary. A non-bit field member that follows a bit field is aligned on the next byte boundary.

## Example of bit field alignment

### Bit-packed example

For:

```
#pragma options align=bit_packed
struct {
    int a : 8;
    int b : 10;
    int c : 12;
    int d : 4;
    int e : 3;
    int : 0;
    int f : 1;
    char g;
} A;
```

```
pragma options align=reset
```

The size of A is 7 bytes. The alignment of A is 1 byte. The layout of A is:

| Member name | Byte offset | Bit offset |
|-------------|-------------|------------|
| a           | 0           | 0          |
| b           | 1           | 0          |
| c           | 2           | 2          |
| d           | 3           | 6          |
| e           | 4           | 2          |
| f           | 5           | 0          |
| g           | 6           | 0          |

---

## Chapter 3. Handling floating point operations

XL C/C++ supports single-precision floating-point numbers with an approximate range of  $10^{-38}$  to  $10^{+38}$ , and about 7 decimal digits of precision; and double-precision floating-point numbers with an approximate range of  $10^{-308}$  to  $10^{+308}$  and precision of about 16 decimal digits. Quadruple precision values have the same range as double precision values, but the precision is about 29 decimal digits.

The following sections provide reference information, portability considerations, and suggested procedures for using compiler options to manage floating-point operations:

- “Handling multiply-add operations”
- “Handling floating-point rounding”
- “Handling floating-point exceptions” on page 12
- “Single-precision and double-precision performance” on page 13
- “Using the Mathematical Acceleration Subsystem (MASS)” on page 13

---

### Handling multiply-add operations

By default, the compiler violates certain IEEE 754 floating-point rules in order to improve performance. For example, multiply-add instructions are generated by default because they are faster and produce a more precise result than separate multiply and add instructions. If you want greater compatibility with the accuracy available on other systems, you can use the **-qfloat=nomaf** option to suppress the generation of these multiply-add instructions.

#### Related references

- **-qfloat** in *XL C/C++ Compiler Reference*

---

### Handling floating-point rounding

By default, the compiler attempts to perform as much arithmetic as possible at compile time. A floating-point operation with constant operands is *folded*, which means that the arithmetical expression is replaced with the compile-time result. If you enable optimization, increased folding might occur. However, the result of a compile-time computation might differ slightly from the result that would have been calculated at run time, because more rounding operations occur at compile time. For example, where a multiply-add-fused (MAF) operation might be used at run time with less rounding, separate multiply and add operations might be used at compile time, producing a slightly different result.

To prevent the possibility of unexpected results due to compile-time rounding, you have two options:

- Use the **-qfloat=nofold** compiler option to suppress all compile-time folding of floating-point computations.
- Use the **-y** compiler option to specify a IEEE compile-time rounding mode that matches the rounding mode to be used at run time. By default, the rounding mode is *round-to-nearest*, unless you specify another value (which you can do via the XL C/C++ built-in function `__setrnd`, declared in the **builtins.h** file).

For example, if you were to compile the following code sample with `-yz`, which specifies a rounding mode of round-to-zero, the two results of `u.x` would be slightly different:

```
int main ()
{
    union uu
    {
        float x;
        int i;
    } u;

    volatile float one, three;

    u.x=1.0/3.0;
    printf("1/3=%8X \n", u.i);

    one=1.0;
    three=3.0;
    u.x=one/three;
    printf ("1/3=%8X \n", u.i);
    return 0;
}
```

This is because the calculation of `1.0/3.0` would be folded at compile-time using round-to-zero rounding, while `one/three` would be calculated at run-time using the default rounding mode of round-to-nearest. (Declaring the variables `one` and `three` as **volatile** suppresses folding by the compiler, even under optimization.) The output of the program would be:

```
1/3=3EAAAAAA
1/3=3EAAAAAB
```

To ensure consistency between compile-time and run-time results in this example, you would compile with the option `-yn` (which is the default).

## Related references

- `-qfloat` in *XL C/C++ Compiler Reference*
- `-y` in *XL C/C++ Compiler Reference*
- `__setrnd` in “Appendix B: Built-in Functions” in the *XL C/C++ Compiler Reference*

---

## Handling floating-point exceptions

By default, invalid operations such as division by zero, division by infinity, overflow, and underflow are ignored at run time. However, you can use the `-qflttrap` option to detect these types of exceptions. In addition, you can add suitable support code to your program to allow program execution to continue after an exception occurs, and to modify the results of operations causing exceptions.

Because, however, floating-point computations involving constants are usually folded at compile time, the potential exceptions that would be produced at run time will not occur. To ensure that the `-qflttrap` option traps all run-time floating-point exceptions, consider using the `-qfloat=nofold` option to suppress all compile-time folding.

## Related references

- `-qfloat` in *XL C/C++ Compiler Reference*
- `-qflttrap` in *XL C/C++ Compiler Reference*

---

## Single-precision and double-precision performance

If you compile your application with the default value of `-qarch=com` option or any of the values `pwr`, `pwr2`, `pwrx`, `pwr2s`, or `p2sc`, only double-precision computations are supported. For these architectures, if you need to convert results to single precision, rounding is applied, based on the rounding mode in effect.

With these architectures, because explicit rounding operations are required, single-precision computations are often slower than double-precision computations. With all other values for `-qarch`, single-precision instructions are used for single-precision operations, and are executed with the same speed as double-precision operations.

For more information about the PowerPC floating-point processor, see the *AIX Assembler Language Reference*.

### Related references

- `-qarch` in *XL C/C++ Compiler Reference*

---

## Using the Mathematical Acceleration Subsystem (MASS)

The XL C/C++ Enterprise Edition for AIX ships the Mathematical Acceleration Subsystem (MASS), a set of libraries of tuned mathematical intrinsic functions that provide improved performance over the corresponding `libm.a` library functions. The accuracy and exception handling might not be identical in MASS functions and `libm.a` functions.

The MASS libraries on AIX consist of a library of scalar functions, described in “Using the scalar library,” and a set of vector libraries tuned for specific architectures, described in “Using the vector libraries” on page 14. “Compiling and linking a program with MASS” on page 17 describes how to compile and link a program that uses the MASS libraries, and how to selectively use the MASS scalar library functions in concert with the regular `libm.a` scalar functions.

### Using the scalar library

The MASS scalar library, `libmass.a`, contains an accelerated set of frequently used math intrinsic functions in the AIX system library `libm.a`. These functions all accept double-precision parameters and return a double-precision result, and are summarized in Table 7. To provide the prototypes for the functions, include `math.h` in your source files.

Table 7. MASS scalar library functions

| Function           | Description   | Prototype                             |
|--------------------|---|---------------------------------------|
| <code>sqrt</code>  | Returns the square root of <code>x</code>                   | <code>double sqrt (double x);</code>  |
| <code>rsqrt</code> | Returns the reciprocal of the square root of <code>x</code> | <code>double rsqrt (double x);</code> |
| <code>exp</code>   | Returns the exponential function of <code>x</code>          | <code>double exp (double x);</code>   |
| <code>log</code>   | Returns the natural logarithm of <code>x</code>             | <code>double log (double x);</code>   |
| <code>sin</code>   | Returns the sine of <code>x</code>                          | <code>double sin (double x);</code>   |
| <code>cos</code>   | Returns the cosine of <code>x</code>                        | <code>double cos (double x);</code>   |
| <code>tan</code>   | Returns the tangent of <code>x</code>                       | <code>double tan (double x);</code>   |
| <code>atan</code>  | Returns the arctangent of <code>x</code>                    | <code>double atan (double x);</code>  |

Table 7. MASS scalar library functions (continued)

| Function | Description                                    | Prototype                          |
|----------|--|------------------------------------|
| atan2    | Returns the arctangent of x/y                  | double atan2 (double x, double y); |
| sinh     | Returns the hyperbolic sine of x               | double sinh (double x);            |
| cosh     | Returns the hyperbolic cosine of x             | double cosh (double x);            |
| tanh     | Returns the hyperbolic tangent of x            | double tanh (double x);            |
| dnint    | Returns the nearest integer to x (as a double) | double dnint (double x);           |
| pow      | Returns x raised to the power y                | double pow (double x, double y);   |

The trigonometric functions (**sin**, **cos**, **tan**) return NaN (Not-a-Number) for large arguments ( $\text{abs}(x) > 2^{50} \cdot \pi$ ).

**Note:** In some cases the MASS functions are not as accurate as **libm.a**, and they might handle edge cases differently (**sqrt(Inf)**, for example).

## Using the vector libraries

The MASS vector libraries are shipped in the following archives:

### **libmassv.a**

The general vector library.

### **libmassvp3.a**

Contains some functions that have been tuned for the POWER3 architecture. The remaining functions are identical to those in **libmassv.a**.

### **libmassvp4.a**

Contains some functions that have been tuned for the POWER4 architecture. The remaining functions are identical to those in **libmassv.a**. If you are using POWER5, this library is the recommended choice.

With the exception of a few functions (described below), all of the functions in **libmassv.a**, **libmassvp3.a**, and **libmassvp4.a** accept three parameters: a double-precision or single-precision vector input parameter; a double-precision or single-precision output parameter; and an integer vector-length parameter. The functions are of the form *function\_name* (*y,x,n*), where *x* is the source vector, *y* is the target vector, and *n* is the vector length. The parameters *y* and *x* are assumed to be double-precision for functions with the prefix **v**, and single-precision for functions with the prefix **vs**. As an example, the following code:

```
#include <massv.h>

double x[500], y[500];
int n;
n = 500;
...
vexp (y, x, &n);
```

outputs a vector *y* of length 500 whose elements are  $\exp(x[i])$ , where  $i=0,\dots,499$ .

The single-precision and double-precision functions contained in the vector libraries are summarized in Table 8 on page 15. To provide the prototypes for the functions, include **massv.h** in your source files. Note that in C and C++ applications, only call by reference is supported, even for scalar arguments.



Table 8. MASS vector library functions

| Double-precision function | Single-precision function | Description  | Double-precision function prototype                                   | Single-precision function prototype                                 |
|---------------------------|---------------------------|--|---|---|
| vacos                     | vsacos                    | Sets $y[i]$ to the arccosine of $x[i]$ , for $i=0,\dots,*n-1$  | void vacos (double $y[]$ , double $x[]$ , int $*n$ );                 | void vsacos (float $y[]$ , float $x[]$ , int $*n$ );                |
| vasin                     | vsasin                    | Sets $y[i]$ to the arcsine of $x[i]$ , for $i=0,\dots,*n-1$  | void vasin (double $y[]$ , double $x[]$ , int $*n$ );                 | void vsasin (float $y[]$ , float $x[]$ , int $*n$ );                |
| vatan2                    | vsatan2                   | Sets $z[i]$ to the arctangent of $x[i]/y[i]$ , for $i=0,\dots,*n-1$  | void vatan2 (double $z[]$ , double $x[]$ , double $y[]$ , int $*n$ ); | void vsatan2 (float $z[]$ , float $x[]$ , float $y[]$ , int $*n$ ); |
| vcos                      | vscos                     | Sets $y[i]$ to the cosine of $x[i]$ , for $i=0,\dots,*n-1$   | void vcos (double $y[]$ , double $x[]$ , int $*n$ );                  | void vscos (float $y[]$ , float $x[]$ , int $*n$ );                 |
| vcosh                     | vscosh                    | Sets $y[i]$ to the hyperbolic cosine of $x[i]$ , for $i=0,\dots,*n-1$  | void vcosh (double $y[]$ , double $x[]$ , int $*n$ );                 | void vscosh (float $y[]$ , float $x[]$ , int $*n$ );                |
| vcosisin <sup>1</sup>     | vscosisin <sup>1</sup>    | Sets the real part of $y[i]$ to the cosine of $x[i]$ and the imaginary part of $y[i]$ to the sine of $x[i]$ , for $i=0,\dots,*n-1$ | void vcosisin (double complex $y[]$ , double $x[]$ , int $*n$ );      | void vscosisin (float complex $y[]$ , float $x[]$ , int $*n$ );     |
| vdint                     |                           | Sets $y[i]$ to the integer truncation of $x[i]$ , for $i=0,\dots,*n-1$   | void vdint (double $y[]$ , double $x[]$ , int $*n$ );                 |   |
| vdiv                      | vsdiv                     | Sets $z[i]$ to $x[i]/y[i]$ , for $i=0,\dots,*n-1$  | void vdiv (double $z[]$ , double $x[]$ , double $y[]$ , int $*n$ );   | void vsdiv (float $z[]$ , float $x[]$ , float $y[]$ , int $*n$ );   |
| vdnint                    |                           | Sets $y[i]$ to the nearest integer to $x[i]$ , for $i=0,\dots,*n-1$  | void vdnint (double $y[]$ , double $x[]$ , int $*n$ );                |   |
| vexp                      | vsexp                     | Sets $y[i]$ to the exponential function of $x[i]$ , for $i=0,\dots,*n-1$   | void vexp (double $y[]$ , double $x[]$ , int $*n$ );                  | void vsexp (float $y[]$ , float $x[]$ , int $*n$ );                 |
| vexpm1                    | vsexpm1                   | Sets $y[i]$ to (the exponential function of $x[i]$ )-1, for $i=0,\dots,*n-1$   | void vexpm1 (double $y[]$ , double $x[]$ , int $*n$ );                | void vsexpm1 (float $y[]$ , float $x[]$ , int $*n$ );               |
| vlog                      | vslog                     | Sets $y[i]$ to the natural logarithm of $x[i]$ , for $i=0,\dots,*n-1$  | void vlog (double $y[]$ , double $x[]$ , int $*n$ );                  | void vslog (float $y[]$ , float $x[]$ , int $*n$ );                 |
| vlog10                    | vslog10                   | Sets $y[i]$ to the base-10 logarithm of $x[i]$ , for $i=0,\dots,*n-1$  | void vlog10 (double $y[]$ , double $x[]$ , int $*n$ );                | void vslog10 (float $y[]$ , float $x[]$ , int $*n$ );               |

Table 8. MASS vector library functions (continued)

|   |          |  |  |  |
|---|----------|--|--|--|
| vlog1p  | vslog1p  | Sets y[i] to the natural logarithm of (x[i]+1), for i=0,...,*n-1               | void vlog1p (double y[], double x[], int *n);              | void vslog1p (float y[], float x[], int *n);             |
| vpow  | vspow    | Sets z[i] to x[i] raised to the power y[i], for i=0,...,*n-1                   | void vpow (double z[], double x[], double y[], int *n);    | void vspow (float z[], float x[], float y[], int *n);    |
| vrec  | vsrec    | Sets y[i] to the reciprocal of x[i], for i=0,...,*n-1                          | void vrec (double y[], double x[], int *n);                | void vsrec (float y[], float x[], int *n);               |
| vrsqrt  | vsrsqrt  | Sets y[i] to the reciprocal of the square root of x[i], for i=0,...,*n-1       | void vrsqrt (double y[], double x[], int *n);              | void vsrsqrt (float y[], float x[], int *n);             |
| vsin  | vssin    | Sets y[i] to the sine of x[i], for i=0,...,*n-1                                | void vsin (double y[], double x[], int *n);                | void vssin (float y[], float x[], int *n);               |
| vsincos   | vssincos | Sets y[i] to the sine of x[i] and z[i] to the cosine of x[i], for i=0,...,*n-1 | void vsincos (double y[], double z[], double x[], int *n); | void vssincos (float y[], float z[], float x[], int *n); |
| vsinh   | vssinh   | Sets y[i] to the hyperbolic sine of x[i], for i=0,...,*n-1                     | void vsinh (double y[], double x[], int *n);               | void vssinh (float y[], float x[], int *n);              |
| vsqrt   | vssqrt   | Sets y[i] to the square root of x[i], for i=0,...,*n-1                         | void vsqrt (double y[], double x[], int *n);               | void vssqrt (float y[], float x[], int *n);              |
| vtan  | vstan    | Sets y[i] to the tangent of x[i], for i=0,...,*n-1                             | void vtan (double y[], double x[], int *n);                | void vstan (float y[], float x[], int *n);               |
| vtanh   | vstanh   | Sets y[i] to the hyperbolic tangent of x[i], for i=0,...,*n-1                  | void vtanh (double y[], double x[], int *n);               | void vstanh (float y[], float x[], int *n);              |
| <p><b>Notes:</b></p> <ol style="list-style-type: none"> <li>By default, these functions use the <b>__Complex</b> data type, which is only available for AIX 5.2 and later, and will not compile on older versions of the operating system. To get an alternate prototype for these functions, compile with <b>-D__nocomplex</b>. This will define the functions as: void vcosisin (double y[2][], double x[], int *n); and void vscosisin (float y[2][], float x[], int *n);</li> </ol> |          |  |  |  |

The functions **vdiv**, **vsincos**, and **vatan2** take four parameters. The functions **vdiv** and **vatan2** take the parameters (z,x,y,n). The function **vdiv** outputs a vector z whose elements are x[i]/y[i], where i=0,...,\*n-1. The function **vatan2** outputs a vector z whose elements are atan(x[i]/y[i]), where i=0,...,\*n-1. The function **vsincos** takes the parameters (y,z,x,n), and outputs two vectors, y and z, whose elements are sin(x[i]) and cos(x[i]) respectively.

In **vcosisin(y,x,n)**, x is a vector of n **double** elements and the function outputs a vector y of n **double complex** elements of the form (cos(x[i]),sin(x[i])). If **-D\_\_nocomplex** is used (see note in 15), the output vector holds y[0][i] = cos(x[i]) and y[1][i] = sin(x[i]), where i=0,...,\*n-1.

## Consistency of MASS vector functions

In the interest of speed, the MASS libraries make certain trade-offs. One of these involves the consistency of certain MASS vector functions. For certain functions, it is possible that the result computed for a particular input value will vary slightly (usually only in the least significant bit) depending on its position in the vector, the vector length, and nearby elements of the input vector. Also, the results produced by the different MASS libraries are not necessarily bit-wise identical.

However, the **libmassvp4.a** library provides newer, consistent versions of certain functions. These consistent functions are: **vsqrt**, **vssqrt**, **vlog**, **vrec**, **vdiv**, **vsin**, **vcos**, **vacos**, **vasin**, **vatan2**, **vrsqrt**, **vscos**, **vsdiv**, **vexp**, **vsrec**, **vssin**.

The accuracy of the vector functions is comparable to that of the corresponding scalar functions in **libmass.a**, though results might not be bit-wise identical.

For more information on consistency and avoiding inconsistency with the vector libraries, as well as performance and accuracy data, see the MASS Web site at <http://www.ibm.com/software/awdtools/vacpp/mass>.

## Related references

- **-D** in *XL C/C++ Compiler Reference*

## Compiling and linking a program with MASS

To compile an application that calls the routines in these libraries, specify **mass** and **massv** (or **massvp3** or **massvp4**) on the **-l** linker option. For example, if the MASS libraries are installed in the default directory **/usr/lib**, you could specify:

```
xlc prog.c -o progf -lmass -lmassv
```

The MASS functions must run in the round-to-nearest rounding mode and with floating-point exception trapping disabled. (These are the default compilation settings.)

## Using libmass.a with libm.a

If you wish to use the **libmass.a** scalar library for some functions and the normal **libm.a** for other functions, follow this procedure to compile and link your program:

1. Create an export list (this can be a flat text file) containing the names of the desired functions. For example, to select only the fast tangent function from **libmass.a** for use with the C program **sample.c**, create a file called **fast\_tan.exp** with the following line:  
tan
2. Create a shared object from the export list with the AIX **ld** command, linking with the **libmass.a** library. For example:  
ld -bexport:fast\_tan.exp -o fast\_tan.o -bnoentry -lmass -bmodtype:SRE
3. Archive the shared object into a library with the AIX **ar** command. For example:  
ar -q libfasttan.a fast\_tan.o
4. Create the final executable using **xlc**, specifying the object file containing the MASS functions *before* the standard math library, **libm.a**. This links only the functions specified in the object file (in this example, the **tan** function) and the remainder of the math functions from the standard system library. For example:  
xlc sample.c -o sample -Ldir\_containing\_libfasttan.a -lfasttan -lm

**Note:** The MASS **cos** function is automatically linked if you export MASS **sin**;  
MASS **atan2** is automatically linked if you export MASS **atan**.

**Related references:**

- **ld** in the *AIX Commands Reference*
- **ar** in the *AIX Commands Reference*

---

## Chapter 4. Using memory heaps

In addition to the memory management functions defined by ANSI, XL C/C++ provides enhanced versions of memory management functions that can help you improve program performance and debug your programs. These functions allow you to:

- Allocate memory from multiple, custom-defined pools of memory, known as user-created heaps.
- Debug memory problems in the default run-time heap.
- Debug memory problems in user-created heaps.

All the versions of the memory management functions actually work in the same way. They differ only in the heap from which they allocate, and in whether they save information to help you debug memory problems. The memory allocated by all of these functions is suitably aligned for storing any type of object.

“Managing memory with multiple heaps” discusses the advantages of using multiple, user-created heaps; summarizes the functions available to manage user-created heaps; provides procedures for creating, expanding, using, and destroying user-defined heaps; and provides examples of programs that create user heaps using both regular and shared memory.

“Debugging memory heaps” on page 30 discusses the functions available for checking and debugging the default and user-created heaps.

---

### Managing memory with multiple heaps

You can use XL C/C++ to create and manipulate your own memory heaps, either in place of or in addition to the default XL C run-time heap.

You can create heaps of regular memory or shared memory, and you can have any number of heaps of any type. The only limit is the space available on your operating system (your machine’s memory and swapper size, minus the memory required by other running applications). You can also change the default run-time heap to a heap that you have created.

Using your own heaps is optional, and your applications will work well using the default memory management provided (and used by) the XL C/C++ run-time library. However, using multiple heaps can be more efficient and can help you improve your program’s performance and reduce wasted memory for a number of reasons:

- When you allocate from a single heap, you can end up with memory blocks on different pages of memory. For example, you might have a linked list that allocates memory each time you add a node to the list. If you allocate memory for other data in between adding nodes, the memory blocks for the nodes could end up on many different pages. To access the data in the list, the system might have to swap many pages, which can significantly slow your program.

With multiple heaps, you can specify the heap from which you want to allocate. For example, you might create a heap specifically for a linked list. The list’s memory blocks and the data they contain would remain close together on fewer pages, which reduces the amount of swapping required.

- In multithreaded applications, only one thread can access the heap at a time to ensure memory is safely allocated and freed. For example, if thread 1 is allocating memory, and thread 2 has a call to **free**, thread 2 must wait until thread 1 has finished its allocation before it can access the heap. Again, this can slow down performance, especially if your program does a lot of memory operations.

If you create a separate heap for each thread, you can allocate from them concurrently, eliminating both the waiting period and the overhead required to serialize access to the heap.

- With a single heap, you must explicitly free each block that you allocate. If you have a linked list that allocates memory for each node, you have to traverse the entire list and free each block individually, which can take some time.

If you create a separate heap for that linked list, you can destroy it with a single call and free all the memory at once.

- When you have only one heap, all components share it (including the XL C/C++ run-time library, vendor libraries, and your own code). If one component corrupts the heap, another component might fail. You might have trouble discovering the cause of the problem and where the heap was damaged.

With multiple heaps, you can create a separate heap for each component, so if one damages the heap (for example, by using a freed pointer), the others can continue unaffected. You also know where to look to correct the problem.

The following sections describe the functions available for using multiple heaps, provide programming guidelines for creating, using and destroying multiple heaps, and provide code examples that implement multiple heaps.

## Functions for managing user-created heaps

The **libhu.a** library provides a set of functions that allow you to manage user-created heaps. These functions are all prefixed by **\_u** (for "user" heaps), and they are declared in the header file **umalloc.h**. The following table summarizes the functions available for creating and managing user-defined heaps.

*Table 9. Functions for managing memory heaps*

| Default heap function | Corresponding user-created heap function | Description  |
|-----------------------|--|--|
| n/a                   | <code>_ucreate</code>                    | Creates a heap. Described in "Creating a heap" on page 21.   |
| n/a                   | <code>_uopen</code>                      | Opens a heap for use by a process. Described in "Using a heap" on page 23.                             |
| n/a                   | <code>_ustats</code>                     | Provides information about a heap. Described in "Getting information about a heap" on page 24.         |
| n/a                   | <code>_uaddmem</code>                    | Adds memory blocks to a heap. Described in "Expanding a heap" on page 22.                              |
| n/a                   | <code>_uclose</code>                     | Closes a heap from further use by a process. Described in "Closing and destroying a heap" on page 24.  |
| n/a                   | <code>_udestroy</code>                   | Destroys a heap. Described in "Closing and destroying a heap" on page 24.                              |
| <code>calloc</code>   | <code>_ucalloc</code>                    | Allocates and initializes memory from a heap you have created. Described in "Using a heap" on page 23. |

Table 9. Functions for managing memory heaps (continued)

| Default heap function | Corresponding user-created heap function | Description  |
|-----------------------|--|--|
| malloc                | _umalloc                                 | Allocates memory from a heap you have created. Described in "Using a heap" on page 23.   |
| _heapmin              | _uheapmin                                | Returns unused memory to the system. Described in "Closing and destroying a heap" on page 24.                                    |
| n/a                   | _udefault                                | Changes the default run-time heap to a user-created heap. Described in "Changing the default heap used in a program" on page 25. |

**Note:** There are no user-created heap versions of **realloc** or **free**. These standard functions always determine the heap from which memory is allocated, and can be used with both user-created and default memory heaps.

## Creating a heap

You can create a fixed-size heap, or a dynamically-sized heap. With a fixed-size heap, the initial block of memory must be large enough to satisfy all allocation requests made to it. With a dynamically-sized heap, the heap can expand and contract as your program needs demand. Procedures for creating both types of heaps are provided below.

### Creating a fixed-size heap

When you create a fixed-size heap, you first allocate a block of memory large enough to hold the heap and to hold internal information required to manage the heap, and you assign it a handle. For example:

```
Heap_t fixedHeap; /* this is the "heap handle" */
/* get memory for internal info plus 5000 bytes for the heap */
static char block[_HEAP_MIN_SIZE + 5000];
```

The internal information requires a minimum set of bytes, specified by the `_HEAP_MIN_SIZE` macro (defined in `umalloc.h`). You can add the amount of memory your program requires to this value to determine the size of the block you need to get. Once the block is fully allocated, further allocation requests to the heap will fail.

After you have allocated a block of memory, you create the heap with `_ucreate`, and specify the type of memory for the heap, regular or shared. For example:

```
fixedHeap = _ucreate(block, (_HEAP_MIN_SIZE+5000), /* block to use */
                    !_BLOCK_CLEAN, /* memory is not set to 0 */
                    _HEAP_REGULAR, /* regular memory */
                    NULL, NULL); /* functions for expanding and shrinking
                                a dynamically-sized heap */
```

The `!_BLOCK_CLEAN` parameter indicates that the memory in the block has not been initialized to 0. If it were set to 0 (for example, by `memset`), you would specify `_BLOCK_CLEAN`. The `calloc` and `ucalloc` functions use this information to improve their efficiency; if the memory is already initialized to 0, they don't need to initialize it.

The fourth parameter indicates the type of memory the heap contains: regular (`_HEAP_REGULAR`) or shared (`_HEAP_SHARED`).

For a fixed-size heap, the last two parameters are always `NULL`.

### Creating a dynamically-sized heap

With the XL C/C++ default heap, when not enough storage is available to fulfill a `malloc` request, the run-time environment gets additional storage from the system. Similarly, when you minimize the heap with `_heapmin` or when your program ends, the run-time environment returns the memory to the operating system.

When you create an expandable heap, you provide your own functions to do this work, which you can name however you choose. You specify pointers to these functions as the last two parameters to `_ucreate` (instead of the `NULL` pointers you use to create a fixed-size heap). For example:

```
Heap_t growHeap;
static char block[_HEAP_MIN_SIZE]; /* get block */

growHeap = _ucreate(block, _HEAP_MIN_SIZE, /* starting block */
                    !_BLOCK_CLEAN,      /* memory not set to 0 */
                    _HEAP_REGULAR,      /* regular memory */
                    expandHeap,         /* function to expand heap */
                    shrinkHeap);       /* function to shrink heap */
```

**Note:** You can use the same expand and shrink functions for more than one heap, as long as the heaps use the same type of memory and your functions are not written specifically for one heap.

## Expanding a heap

To increase the size of a heap, you add blocks of memory to it by doing the following:

- For fixed-size or dynamically-sized heaps, calling the `_uaddmem` function.
- For dynamically-sized heaps only, writing a function that expands the heap, and that can be called automatically by the system if necessary, whenever you allocate memory from the heap.

Both options are described below.

### Adding blocks of memory to a heap

You can add blocks of memory to a fixed-size or dynamically-sized heap with `_uaddmem`. This can be useful if you have a large amount of memory that is allocated conditionally. Like the starting block, you must first allocate memory for a block of memory. This block will be added to the current heap, so make sure the block you add is of the same type of memory as the heap to which you are adding it. For example, to add 64K to `fixedHeap`:

```
static char newblock[65536];

_uaddmem(fixedHeap, /* heap to add to */
         newblock, 65536, /* block to add */
         _BLOCK_CLEAN); /* sets memory to 0 */
```

**Note:** For every block of memory you add, a small number of bytes from it are used to store internal information. To reduce the total amount of overhead, it is better to add a few large blocks of memory than many small blocks.

### Writing a heap-expanding function

When you call `_umalloc` (or a similar function) for a dynamically-sized heap, `_umalloc` tries to allocate the memory from the initial block you provided to `_ucreate`. If not enough memory is there, it then calls the heap-expanding function



you specified as a parameter to `_ucreate`. Your function then gets more memory from the operating system and adds it to the heap. It is up to you how you do this.

Your function must have the following prototype:

```
void *(*functionName)(Heap_t uh, size_t *size, int *clean);
```

Where *functionName* identifies the function (you can name it however you want), *uh* is the heap to be expanded, and *size* is the size of the allocation request passed by `_umalloc`. You probably want to return enough memory at a time to satisfy several allocations; otherwise every subsequent allocation has to call your heap-expanding function, reducing your program's execution speed. Make sure that you update the *size* parameter if you return more than the *size* requested.

Your function must also set the *clean* parameter to either `_BLOCK_CLEAN`, to indicate the memory has been set to 0, or `!_BLOCK_CLEAN`, to indicate that the memory has not been initialized.

The following fragment shows an example of a heap-expanding function:

```
static void *expandHeap(Heap_t uh, size_t *length, int *clean)
{
    char *newblock;
    /* round the size up to a multiple of 64K * /
    *length = (*length / 65536) * 65536 + 65536;

    *clean = _BLOCK_CLEAN; /* mark the block as "clean" */
    return(newblock);      /* return new memory block */
}
```

## Using a heap

Once you have created a heap, you can open it for use by calling `_uopen`:

```
_uopen(fixedHeap);
```

This opens the heap for that particular process; if the heap is shared, each process that uses the heap needs its own call to `_uopen`.

You can then allocate and free memory from your own heap just as you would from the default heap. To allocate memory, use `_ucalloc` or `_umalloc`. These functions work just like `calloc` and `malloc`, except you specify the heap to use as well as the size of block that you want. For example, to allocate 1000 bytes from `fixedHeap`:

```
void *up;
up = _umalloc(fixedHeap, 1000);
```

To reallocate and free memory, use the regular `realloc` and `free` functions. Both of these functions always check the heap from which the memory was allocated, so you don't need to specify the heap to use. For example, the `realloc` and `free` calls in the following code fragment look exactly the same for both the default heap and your heap:

```
void *p, *up;
p = malloc(1000); /* allocate 1000 bytes from default heap */
up = _umalloc(fixedHeap, 1000); /* allocate 1000 from fixedHeap */

realloc(p, 2000); /* reallocate from default heap */
realloc(up, 100); /* reallocate from fixedHeap */
```

```
free(p);          /* free memory back to default heap */
free(up);        /* free memory back to fixedHeap  */
```

When you call any heap function, make sure the heap you specify is valid. If the heap is not valid, the behavior of the heap functions is undefined.

## Getting information about a heap

You can determine the heap from which any object was allocated by calling `_mheap`. You can also get information about the heap itself by calling `_ustats`, which tells you:

- The amount of memory the heap holds (excluding memory used for overhead)
- The amount of memory currently allocated from the heap
- The type of memory in the heap
- The size of the largest contiguous piece of memory available from the heap

## Closing and destroying a heap

When a process has finished using the heap, close it with `_uclose`. Once you have closed the heap in a process, that process can no longer allocate from or return memory to that heap. If other processes share the heap, they can still use it until you close it in each of them. Performing operations on a heap after you have closed it causes undefined behavior.

To destroy a heap, do the following:

- For a fixed-size heap, call `_udestroy`. If blocks of memory are still allocated somewhere, you can force the destruction. Destroying a heap removes it entirely even if it was shared by other processes. Again, performing operations on a heap after you have destroyed it causes undefined behavior.
- For a dynamically-sized heap, call `_uheapmin` to coalesce the heap (return all blocks in the heap that are totally free to the system), or `_udestroy` to destroy it. Both of these functions call your heap-shrinking function. (See below.)

After you destroy a heap, it is up to you to return the memory for the heap (the initial block of memory you supplied to `_ucreate` and any other blocks added by `_uaddmem`) to the system.

## Writing the heap-shrinking function

When you call `_uheapmin` or `_udestroy` to coalesce or destroy a dynamically-sized heap, these functions call your heap-shrinking function to return the memory to the system. It is up to you how you implement this function.

Your function must have the following prototype:

```
void (*functionName)(Heap_t uh, void *block, size_t size);
```

Where *functionName* identifies the function (you can name it however you want), *uh* identifies the heap to be shrunk. The pointer *block* and its *size* are passed to your function by `_uheapmin` or `_udestroy`. Your function must return the memory pointed to by *block* to the system. For example:

```
static void shrinkHeap(Heap_t uh, void *block, size_t size)
{
    free(block);
    return;
}
```

## Changing the default heap used in a program

The regular memory management functions (**malloc** and so on) always use the current default heap for that thread. The initial default heap for all XL C/C++ applications is the run-time heap provided by XL C/C++. However, you can make your own heap the default by calling **\_udefault**. Then all calls to the regular memory management functions allocate memory from your heap instead of the default run-time heap.

The default heap changes only for the thread where you call **\_udefault**. You can use a different default heap for each thread of your program if you choose. This is useful when you want a component (such as a vendor library) to use a heap other than the XL C/C++ default heap, but you cannot actually alter the source code to use heap-specific calls. For example, if you set the default heap to a shared heap and then call a library function that calls **malloc**, the library allocates storage in shared memory.

Because **\_udefault** returns the current default heap, you can save the return value and later use it to restore the default heap you replaced. You can also change the default back to the XL C/C++ default run-time heap by calling **\_udefault** and specifying the **\_RUNTIME\_HEAP** macro (defined in **umalloc.h**). You can also use this macro with any of the heap-specific functions to explicitly allocate from the default run-time heap.

## Compiling and linking a program with user-created heaps

To compile an application that calls any of the user-created heap functions (prefixed by **\_u**), specify **hu** on the **-l** linker option. For example, if the **libhu.a** library is installed in the default directory, you could specify:

```
xlc prog.c -o progf -lhu
```

## Examples of creating and using user heaps

### Example of a user heap with regular memory

The program below shows how you might create and use a heap that uses regular memory.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>

static void *get_fn(Heap_t usrheap, size_t *length, int *clean)
{
    void *p;
    /* Round up to the next chunk size */
    *length = ((*length) / 65536) * 65536 + 65536;
    *clean = _BLOCK_CLEAN;
    p = calloc(*length,1);
    return (p);
}

static void release_fn(Heap_t usrheap, void *p, size_t size)
{
    free( p );
    return;
}

int main(void)
{
    void    *initial_block;
    long    rc;
```

```

Heap_t myheap;
char *ptr;
int initial_sz;

/* Get initial area to start heap */
initial_sz = 65536;
initial_block = malloc(initial_sz);
if(initial_block == NULL) return (1);

/* create a user heap */
myheap = _ucreate(initial_block, initial_sz, _BLOCK_CLEAN,
                 _HEAP_REGULAR, get_fn, release_fn);
if (myheap == NULL) return(2);

/* allocate from user heap and cause it to grow */
ptr = _umalloc(myheap, 100000);
_free(ptr);

/* destroy user heap */
if (_udestroy(myheap, _FORCE)) return(3);

/* return initial block used to create heap */

free(initial_block);
return 0;
}

```

## Example of a shared user heap – parent process

The following program shows how you might implement a heap shared between a parent and several child processes. This program shows the parent process, which creates the shared heap. First the main program calls the **init** function to allocate shared memory from the operating system (using **CreateFileMapping**) and name the memory so that other processes can use it by name. The **init** function then creates and opens the heap. The loop in the main program performs operations on the heap, and also starts other processes. The program then calls the **term** function to close and destroy the heap.

```

#include <umalloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define PAGING_FILE 0xFFFFFFFF
#define MEMORY_SIZE 65536
#define BASE_MEM (VOID*)0x01000000

static HANDLE hFile; /* Handle to memory file */
static void* hMap; /* Handle to allocated memory */

typedef struct mem_info {
    void * pBase;
    Heap_t pHeap;
} MEM_INFO_T;

/*-----*/
/* inithp: */
/* Function to create and open the heap with a named shared memory object */
/*-----*/
static Heap_t inithp(size_t heap_size)
{
    MEM_INFO_T info; /* Info structure */

    /* Allocate shared memory from the system by creating a shared memory */
    /* pool basing it out of the system paging (swapper) file. */
}

```

```

hFile = CreateFileMapping( (HANDLE) PAGING_FILE, NULL, PAGE_READWRITE, 0,
                          heap_size + sizeof(Heap_t), "MYNAME_SHAREMEM" );
if (hFile == NULL) {
    return NULL;
}

/* Map the file to this process' address space, starting at an address */
/* that should also be available in child processe(s) */

hMap = MapViewOfFileEx( hFile, FILE_MAP_WRITE, 0, 0, 0, BASE_MEM );

info.pBase = hMap;
if (info.pBase == NULL) {
    return NULL;
}

/* Create a fixed sized heap. Put the heap handle as well as the */
/* base heap address at the beginning of the shared memory. */

info.pHeap = _ucreate((char *)info.pBase + sizeof(info), heap_size - sizeof(info),
                     !_BLOCK_CLEAN, _HEAP_SHARED | _HEAP_REGULAR, NULL, NULL);

if (info.pBase == NULL) {
    return NULL;
}

memcpy(info.pBase, info, sizeof(info));

if (_uopen(info.pHeap)) { /* Open heap and check result */
    return NULL;
}

return info.pHeap;
}

/*-----*/
/* termhp: */
/* Function to close and destroy the heap */
/*-----*/
static int termhp(Heap_t uheap)
{
    if (_uclose(uheap)) /* close heap */
        return 1;
    if (_udestroy(uheap, _FORCE)) /* force destruction of heap */
        return 1;

    UnmapViewOfFile(hMap); /* return memory to system */
    CloseHandle(hFile);

    return 0;
}

/*-----*/
/* main: */
/* Main function to test creating, writing to and destroying a shared */
/* heap. */
/*-----*/
int main(void)
{
    int i, rc; /* Index and return code */
    Heap_t uheap; /* heap to create */
    char *p; /* for allocating from heap */

    /*
    /* call init function to create and open the heap

```

```

uheap = inithp(MEMORY_SIZE);
if (uheap == NULL)           /* check for success      */
    return 1;                 /* if failure, return non zero */

/*                               */
/* perform operations on uheap  */
/*                               */
for (i = 1; i <= 5; i++)
{
    p = _umalloc(uheap, 10);   /* allocate from uheap      */
    if (p == NULL)
        return 1;
    memset(p, 'M', _msize(p)); /* set all bytes in p to 'M' */
    p = realloc(p,50);        /* reallocate from uheap    */
    if (p == NULL)
        return 1;
    memset(p, 'R', _msize(p)); /* set all bytes in p to 'R' */
}

/*                               */
/* Start a second process which accesses the heap */
/*                               */
if (system("memshr2.exe"))
    return 1;

/*                               */
/* Take a look at the memory that we just wrote to. Note that memshr.c */
/* and memshr2.c should have been compiled specifying the */
/* alloc(debug[, yes]) flag. */
/*                               */
#ifdef DEBUG
    _udump_allocated(uheap, -1);
#endif

/*                               */
/* call term function to close and destroy the heap */
/*                               */
rc = termhp(uheap);

#ifdef DEBUG
    printf("memshr ending... rc = %d\n", rc);
#endif

return rc;
}

```

### Example of a shared user heap - child process

The following program shows the process started by the loop in the parent process. This process uses **OpenFileMapping** to access the shared memory by name, then extracts the heap handle for the heap created by the parent process. The process then opens the heap, makes it the default heap, and performs some operations on it in the loop. After the loop, the process replaces the old default heap, closes the user heap, and ends.

```

#include <umalloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static HANDLE hFile;           /* Handle to memory file      */
static void* hMap;            /* Handle to allocated memory */

typedef struct mem_info {
    void * pBase;

```

```

    Heap_t pHeap;
} MEM_INFO_T;

/*-----*/
/* inithp: Subprocess Version */
/* Function to create and open the heap with a named shared memory object */
/*-----*/
static Heap_t inithp(void)
{
    MEM_INFO_T info;                /* Info structure */

    /* Open the shared memory file by name. The file is based on the
    /* system paging (swapper) file.

    hFile = OpenFileMapping(FILE_MAP_WRITE, FALSE, "MYNAME_SHAREMEM");

    if (hFile == NULL) {
        return NULL;
    }

    /* Figure out where to map this file by looking at the address in the
    /* shared memory where the memory was mapped in the parent process.

    hMap = MapViewOfFile( hFile, FILE_MAP_WRITE, 0, 0, sizeof(info) );

    if (hMap == NULL) {
        return NULL;
    }

    /* Extract the heap and base memory address from shared memory

    memcpy(info, hMap, sizeof(info));
    UnmapViewOfFile(hMap);

    hMap = MapViewOfFileEx( hFile, FILE_MAP_WRITE, 0, 0, 0, info.pBase );

    if (_uopen(info.pHeap)) {        /* Open heap and check result */
        return NULL;
    }

    return info.pHeap;
}

/*-----*/
/* termhp: */
/* Function to close my view of the heap */
/*-----*/
static int termhp(Heap_t uheap)
{
    if (_uclose(uheap))              /* close heap */

    UnmapViewOfFile(hMap);          /* return memory to system */
    CloseHandle(hFile);

    return 0;
}

/*-----*/
/* main: */
/* Main function to test creating, writing to and destroying a shared
/* heap.
/*-----*/
int main(void)
{
    int rc, i;                      /* for return code, loop iteration */

```

```

Heap_t uheap, oldheap;          /* heap to create, old default heap */
char *p;                       /* for allocating from the heap */

/*                               */
/* Get the heap storage from the shared memory */
/*                               */
uheap = inithp();
if (uheap == NULL)
    return 1;

/*                               */
/* Register uheap as default run-time heap, save old default */
/*                               */
oldheap = _udefault(uheap);
if (oldheap == NULL) {
    return termhp(uheap);
}

/*                               */
/* Perform operations on uheap */
/*                               */
for (i = 1; i <= 5; i++)
{
    p = malloc(10);          /* malloc uses default heap, which is now uheap*/
    memset(p, 'M', _msize(p));
}

/*                               */
/* Replace original default heap and check result */
/*                               */
if (uheap != _udefault(oldheap)) {
    return termhp(uheap);
}

/*                               */
/* Close my views of the heap */
/*                               */
rc = termhp(uheap);

#ifdef DEBUG
    printf("Returning from memshr2 rc = %d\n", rc);
#endif
return rc;
}

```

---

## Debugging memory heaps

XL C/C++ provides two sets of functions for debugging memory problems:

- Heap-checking functions similar to those provided by other compilers. (Described in “Functions for checking memory heaps” on page 31.)
- Debug versions of all memory management functions. (Described in “Functions for debugging memory heaps” on page 31.)

Both sets of debugging functions have their benefits and drawbacks. The one you choose to use depends on your program, your problems, and your preference.

The heap-checking functions perform more general checks on the heap at specific points in your program. You have greater control over where the checks occur. The heap-checking functions also provide compatibility with other compilers that offer these functions. You only have to rebuild the modules that contain the heap-checking calls. However, you have to change your source code to include these calls, which you will probably want to remove in your final code. Also, the



heap-checking functions only tell you if the heap is consistent or not; they do not provide the details that the debug memory management functions do.

On the other hand, the debug memory management functions provide detailed information about all allocation requests you make with them in your program. You don't need to change any code to use the debug versions; you need only specify the `-qheapdebug` option.

A recommended approach is to add calls to heap-checking functions in places you suspect possible memory problems. If the heap turns out to be corrupted, you can rebuild with `-qheapdebug`.

Regardless of which debugging functions you choose, your program requires additional memory to maintain internal information for these functions. If you are using fixed-size heaps, you might have to increase the heap size in order to use the debugging functions.

## Related references

- “Memory debug library functions,” on page 79

## Functions for checking memory heaps

The header file `umalloc.h` declares a set of functions for validating user-created heaps. These functions are not controlled by a compiler option, so you can use them in your program at any time. Regular versions of these functions, without the `_u` prefix, are also available for checking the default heap. The heap-checking functions are summarized in the following table.

Table 10. Functions for checking memory heaps

| Default heap function   | User-created heap function | Description   |
|-------------------------|----------------------------|---|
| <code>_heapchk</code>   | <code>_uheapchk</code>     | Checks the entire heap for minimal consistency.   |
| <code>_heapset</code>   | <code>_uheapset</code>     | Checks the free memory in the heap for minimal consistency, and sets the free memory in the heap to a value you specify.  |
| <code>_heap_walk</code> | <code>_uheap_walk</code>   | Traverses the heap and provides information about each allocated or freed object to a callback function that you provide. |

To compile an application that calls the user-created heap functions, see “Compiling and linking a program with user-created heaps” on page 25.

## Functions for debugging memory heaps

Debug versions are available for both regular memory management functions and user-defined heap memory management functions. Each debug version performs the same function as its non-debug counterpart, and you can use them for any type of heap, including shared memory. Each call you make to a debug function also automatically checks the heap by calling `_heap_check` (described below), and provides information, including file name and line number, that you can use to debug memory problems. The names of the user-defined debug versions are prefixed by `_debug_u` (for example, `_debug_umalloc`), and they are defined in `umalloc.h`.

For a complete list and details about all of the debug memory management functions, see “Memory debug library functions,” on page 79.

Table 11. Functions for debugging memory heaps

| Default heap function       | Corresponding user-created heap function |
|-----------------------------|--|
| <code>_debug_calloc</code>  | <code>_debug_ucalloc</code>              |
| <code>_debug_malloc</code>  | <code>_debug_umalloc</code>              |
| <code>_debug_heapmin</code> | <code>_debug_uheapmin</code>             |
| <code>_debug_realloc</code> | n/a                                      |
| <code>_debug_free</code>    | n/a                                      |

To use these debug versions, you can do either of the following:

- In your source code, prefix any of the default or user-defined-heap memory management functions with `_debug_`.
- If you do not wish to make changes to the source code, simply compile with the `-qheapdebug` option. This option maps all calls to memory management functions to their debug version counterparts. To prevent a call from being mapped, parenthesize the function name.

To compile an application that calls the user-created heap functions, see “Compiling and linking a program with user-created heaps” on page 25.

**Notes:**

1. When the `-qheapdebug` option is specified, code is generated to *pre-initialize* the local variables for all functions. This makes it much more likely that uninitialized local variables will be found during the normal debug cycle rather than much later (usually when the code is optimized).
2. Do not use the `-brtl` option with `-qheapdebug`.
3. You should place a `#pragma strings (readonly)` directive at the top of each source file that will call debug functions, or in a common header file that each includes. This directive is not essential, but it ensures that the file name passed to the debug functions cannot be overwritten, and that only one copy of the file name string is included in the object module.

**Additional functions for debugging memory heaps**

Three additional debug memory management functions do not have regular counterparts. They are summarized in the following table.

Table 12. Additional functions for debugging memory heaps

| Default heap function              | Corresponding user-created heap function | Description  |
|------------------------------------|--|--|
| <code>_dump_allocated</code>       | <code>_udump_allocated</code>            | Prints information to <code>stderr</code> about each memory block currently allocated by the debug functions.  |
| <code>_dump_allocated_delta</code> | <code>_udump_allocated_delta</code>      | Prints information to file descriptor 2 about each memory block allocated by the debug functions since the last call to <code>_dump_allocated</code> or <code>_dump_allocated_delta</code> . |

Table 12. Additional functions for debugging memory heaps (continued)

| Default heap function    | Corresponding user-created heap function | Description  |
|--------------------------|--|--|
| <code>_heap_check</code> | <code>_uheap_check</code>                | Checks all memory blocks allocated or freed by the debug functions to make sure that no overwriting has occurred outside the bounds of allocated blocks or in a free memory block. |

The `_heap_check` function is automatically called by the debug functions; you can also call this function explicitly. You can then use `_dump_allocated` or `_dump_allocated_delta` to display information about currently allocated memory blocks. You must explicitly call these functions.

## Using memory allocation fill patterns

Some debug functions set all the memory they allocate to a specified fill pattern. This lets you easily locate areas in memory that your program uses.

The `debug_malloc`, `debug_realloc`, and `debug_umalloc` functions set allocated memory to a default repeating `0xAA` fill pattern. To enable this fill pattern, export the `HD_FILL` environment variable.

The `debug_free` function sets all free memory to a repeating `0xFB` fill pattern.

## Skipping heap checking

Each debug function calls `_heap_check` (or `_uheap_check`) to check the heap. Although this is useful, it can also increase your program's memory requirements and decrease its execution speed.

To reduce the overhead of checking the heap on every debug memory management function, you can control how often the functions check the heap with the `HD_SKIP` environment variable. You will not need to do this for most of your applications unless the application is extremely memory intensive.

Set `HD_SKIP` like any other environment variable. The syntax for `HD_SKIP` is:

```
set HD_SKIP=increment, [start]
```

where:

*increment* Specifies the number of debug function calls to skip between performing heap checks.

*start* Specifies the number debug function calls to skip before starting heap checks.

**Note:** The comma separating the parameters is optional.

For example, if you specify:

```
set HD_SKIP=10
```

then every tenth debug memory function call performs a heap check. If you specify:

```
set HD_SKIP=5,100
```

then after 100 debug memory function calls, only every fifth call performs a heap check.

When you use the *start* parameter to start skipping heap checks, you are trading off heap checks that are done implicitly against program execution speed. You should therefore start with a small increment (like 5) and slowly increase until the application is usable.

## Using stack traces

Stack contents are traced for each allocated memory object. If the contents of an object's stack change, the traced contents are dumped.

The trace size is controlled by the **HD\_STACK** environment variable. If this variable is not set, the compiler assumes a stack size of 10. To disable stack tracing, set the **HD\_STACK** environment variable to 0.

---

## Chapter 5. Using C++ templates

In C++, you can use a template to declare a set of related:

- Classes (including structures)
- Functions
- Static data members of template classes

Within an application, you can instantiate the same template multiple times with the same arguments or with different arguments. If you use the same arguments, the repeated instantiations are redundant. These redundant instantiations increase compilation time, increase the size of the executable, and deliver no benefit.

There are four basic approaches to the problem of redundant instantiations:

### Code for unique instantiations

Organize your source code so that the object files contain only one instance of each required instantiation and no unused instantiations. This is the least usable approach, because you must know where each template is defined and where each template instantiation is required.

### Instantiate at every occurrence

Use the **-qnotempinc** and **-qnotemplateregistry** compiler options (these are the default settings). The compiler generates code for every instantiation that it encounters. With this approach, you accept the disadvantages of redundant instantiations.

### Have the compiler store instantiations in a template include directory

Use the **-qtempinc** compiler option. If the template definition and implementation files have the required structure, each template instantiation is stored in a template include directory. If the compiler is asked to instantiate the same template again with the same arguments, it uses the stored version instead. This approach is described in “Using the **-qtempinc** compiler option.”

### Have the compiler store instantiation information in a registry

Use the **-qtemplateregistry** compiler option. Information about each template instantiation is stored in a template registry. If the compiler is asked to instantiate the same template again with the same arguments, it points to the instantiation in the first object file instead. The **-qtemplateregistry** compiler option provides the benefits of the **-qtempinc** compiler option but does not require a specific structure for the template definition and implementation files. This approach is described in “Using the **-qtemplateregistry** compiler option” on page 38.

**Note:** The **-qtempinc** and **-qtemplateregistry** compiler options are mutually exclusive.

---

## Using the **-qtempinc** compiler option

To use **-qtempinc**, you must structure your application as follows:

- Declare your class templates and function templates in template header files, with a **.h** extension.

- For each template declaration file, create a template implementation file. This file must have the same file name as the template declaration file and an extension of `.c` or `.t`, or the name must be specified in a **#pragma implementation** directive. For a class template, the implementation file defines the member functions and static data members. For a function template, the implementation file defines the function.
- In your source program, specify an **#include** directive for each template declaration file.
- Optionally, to ensure that your code is applicable for both **-qtempinc** and **-qnotempinc** compilations, in each template declaration file, conditionally include the corresponding template implementation file if the `__TEMPINC__` macro is *not* defined. (This macro is automatically defined when you use the **-qtempinc** compilation option.)

This produces the following results:

- Whenever you compile with **-qnotempinc**, the template implementation file is included.
- Whenever you compile with **-qtempinc**, the compiler does not include the template implementation file. Instead, the compiler looks for a file with the same name as the template implementation file and extension `.c` the first time it needs a particular instantiation. If the compiler subsequently needs the same instantiation, it uses the copy stored in the template include directory.

## Example of -qtempinc

This example includes the following source files:

- A template declaration file: `stack.h`.
- The corresponding template implementation file: `stack.c`.
- A function prototype: `stackops.h` (not a function template).
- The corresponding function implementation file: `stackops.cpp`.
- The main program source file: `stackadd.cpp`.

In this example:

1. Both source files include the template declaration file `stack.h`.
2. Both source files include the function prototype `stackops.h`.
3. The template declaration file conditionally includes the template implementation file `stack.c` if the program is compiled with **-qnotempinc**.

### Template declaration file: `stack.h`

This header file defines the class template for the class `Stack`.

```
#ifndef STACK_H
#define STACK_H

template <class Item, int size> class Stack {
public:
    void push(Item item); // Push operator
    Item pop();          // Pop operator
    int isEmpty(){
        return (top==0); // Returns true if empty, otherwise false
    }
    Stack() { top = 0; } // Constructor defined inline
private:
    Item stack[size]; // The stack of items
    int top;          // Index to top of stack
};
```

```

#ifndef __USE_STL_TEMPINC__ // 3
#include "stack.c" // 3
#endif // 3
#endif

```

### Template implementation file: stack.c

This file provides the implementation of the class template for the class Stack.

```

template <class Item, int size>
void Stack<Item,size>::push(Item item) {
    if (top >= size) throw size;
    stack[top++] = item;
}
template <class Item, int size>
Item Stack<Item,size>::pop() {
    if (top <= 0) throw size;
    Item item = stack[--top];
    return(item);
}

```

### Function declaration file: stackops.h

This header file contains the prototype for the add function, which is used in both stackadd.cpp and stackops.cpp.

```
void add(Stack<int, 50>& s);
```

### Function implementation file: stackops.cpp

This file provides the implementation of the add function, which is called from the main program.

```

#include "stack.h" // 1
#include "stackops.h" // 2

void add(Stack<int, 50>& s) {
    int tot = s.pop() + s.pop();
    s.push(tot);
    return;
}

```

### Main program file: stackadd.cpp

This file creates a Stack object.

```

#include <iostream.h>
#include "stack.h" // 1
#include "stackops.h" // 2

main() {
    Stack<int, 50> s; // create a stack of ints
    int left=10, right=20;
    int sum;

    s.push(left); // push 10 on the stack
    s.push(right); // push 20 on the stack
    add(s); // pop the 2 numbers off the stack
            // and push the sum onto the stack
    sum = s.pop(); // pop the sum off the stack

    cout << "The sum of: " << left << " and: " << right << " is: " << sum << endl;

    return(0);
}

```

## Regenerating the template instantiation file

The compiler builds a template instantiation file in the **TEMPINC** directory corresponding to each template implementation file. With each compilation, the compiler can add information to the file but it never removes information from the file.

As you develop your program, you might remove template function references or reorganize your program so that the template instantiation files become obsolete. You can periodically delete the **TEMPINC** destination and recompile your program.

## Using **-qtempinc** with shared libraries

In a traditional application development environment, different applications can share both source files and compiled files. When you use templates, applications can share source files but cannot share compiled files.

If you use **-qtempinc**:

- Each application must have its own **TEMPINC** destination.
- You must compile all of the source files for the application, even if some of the files have already been compiled for another application.

## Related references

- **-qtempinc** in *XL C/C++ Compiler Reference*
- **#pragma implementation** in *XL C/C++ Compiler Reference*

---

## Using the **-qtemplateregistry** compiler option

Unlike **-qtempinc**, the **-qtemplateregistry** compiler option does not impose specific requirements on the organization of your source code. Any program that compiles successfully with **-qnotempinc** will compile with **-qtemplateregistry**.

The template registry uses a "first-come first-served" algorithm:

- When a program references a new instantiation for the first time, it is instantiated in the compilation unit in which it occurs.
- When another compilation unit references the same instantiation, it is not instantiated. Thus, only one copy is generated for the entire program.

The instantiation information is stored in a template registry file. You must use the same template registry file for the entire program. Two programs cannot share a template registry file.

The default file name for the template registry file is **templateregistry**, but you can specify any other valid file name to override this default. When cleaning your program build environment before starting a fresh or scratch build, you must delete the registry file along with the old object files.

## Recompiling related compilation units

If two compilation units, A and B, reference the same instantiation, the **-qtemplateregistry** compiler option has the following effect:

- If you compile A first, the object file for A contains the code for the instantiation.
- When you later compile B, the object file for B does not contain the code for the instantiation because object A already does.



- If you later change A so that it no longer references this instantiation, the reference in object B would produce an unresolved symbol error. When you recompile A, the compiler detects this problem and handles it as follows:
  - If the **-qtemplaterecompile** compiler option is in effect, the compiler automatically recompiles B during the link step, using the same compiler options that were specified for A. (Note, however, that if you use separate compilation and linkage steps, you need to include the compilation options in the link step to ensure the correct compilation of B.)
  - If the **-qnotemplaterecompile** compiler option is in effect, the compiler issues a warning and you must manually recompile B.

## Switching from **-qtempinc** to **-qtemplateregistry**

Because the **-qtemplateregistry** compiler option does not impose any restrictions on the file structure of your application, it has less administrative overhead than **-qtempinc**. You can make the switch as follows:

- If your application compiles successfully with both **-qtempinc** and **-qnotempinc**, you do not need to make any changes.
- If your application compiles successfully with **-qtempinc** but not with **-qnotempinc**, you must change it so that it will compile successfully with **-qnotempinc**. In each template definition file, conditionally include the corresponding template implementation file if the `__TEMPINC__` macro is not defined. This is illustrated in “Example of **-qtempinc**” on page 36.

## Related references

- **-qtemplateregistry** in *XL C/C++ Compiler Reference*
- **-qtemplaterecompile** in *XL C/C++ Compiler Reference*



---

## Chapter 6. Ensuring thread safety (C++)

If you are building multithreaded C++ applications, there are some thread-safety issues which you need to consider when using objects defined in the C++ Standard Template Library and in the stream classes.

---

### Ensuring thread safety of template objects

The following headers in the Standard Template Library are reentrant:

- **algorithm**
- **deque**
- **functional**
- **iterator**
- **list**
- **map**
- **memory**
- **numeric**
- **queue**
- **set**
- **stack**
- **string**
- **unordered\_map**
- **unordered\_set**
- **utility**
- **valarray**
- **vector**

XL C/C++ supports reentrancy to the extent that you can safely read a single object from multiple threads simultaneously. This level of reentrancy is intrinsic. No locks or other globally allocated resources are used.

However, the headers are not reentrant in these cases:

- A single container object is written by multiple threads simultaneously.
- A single container object is written in one thread, while being read in one or more other threads.

If multiple threads write to a single container, or a single thread writes to a single container while other threads are reading from that container, it is your responsibility to serialize access to this container. If multiple threads read from a single container, and no processes write to the container, no serialization is necessary.

---

### Ensuring thread safety of stream objects

All classes declared in the **iostream** standard library are reentrant, and use a single lock to ensure thread-safety while preventing deadlock from occurring. However, on multiprocessor machines, there is a chance, although rare, that livelock can occur when two different threads attempt to concurrently access a shared stream object, or when a stream object holds a lock while waiting for input (for example, from the keyboard). If you want to avoid the possibility of livelock, you can disable locking in input stream objects, output stream objects, or both, by using the following macros at compile time:

`__NOLOCK_ON_INPUT`

Disables input locking.

`__NOLOCK_ON_OUTPUT`

Disables output locking.

To use one or both of these macros, prefix the macro name with the `-D` option on the compilation command line. For example:

```
xlc_r -D__NOLOCK_ON_INPUT -D__NOLOCK_ON_OUTPUT a.C
```

However, if you disable locking on input or output objects, *it is your responsibility* to provide the appropriate locking mechanisms in your source code if stream objects are shared between threads. If you do not, the behavior is undefined, with the possibility of data corruption or application crash.

**Note:** If you use OpenMP directives or the `-qsmp` option to automatically parallelize code which shares input/output stream objects, in conjunction with the lock-disabling macros, you run the same risks as with code that implements Pthreads or other multithreading constructs, and you will need to synchronize the threads accordingly.

## Related references

- `-D` in *XL C/C++ Compiler Reference*
- `-qsmp` in *XL C/C++ Compiler Reference*

---

## Chapter 7. Constructing a library

You can include static and shared libraries in your C and C++ applications.

“Compiling and linking a library” describes how to compile your source files into object files for inclusion in a library, how to link a library into the main program, and how to link one library into another.

“Initializing static objects in libraries (C++)” on page 45 describes how to use *priorities* to control the order of initialization of objects across multiple files in a C++ application.

“Dynamically loading a shared library” on page 49 describes two functions you can use in your application code to load, initialize, unload, and terminate a C++ shared library at run time.

---

### Compiling and linking a library

#### Compiling a static library

To compile a static (unshared) library:

1. Compile each source file into an object file, with no linking.
2. Use the AIX **ar** command to add the generated object files to an archive library file.

For example:

```
xlc -c bar.c example.c
ar -rv libfoo.a bar.o example.o
```

#### Compiling a shared library

To create a shared library that uses static linking:

1. Compile each source file into an object file, with no linking. For example:  

```
xlc -c foo.c -o foo.o
```
2. Optionally, create an export file listing the global symbols to be exported, by doing one of the following:
  - Use the **CreateExportList** utility, described in “Exporting symbols with the CreateExportList utility” on page 44.
  - Use the **-qexpfile=** compiler option with the **-qmkshrobj** option, to create the basis for the export file used in the real link step. For example:  

```
xlc -qmkshrobj -qexpfile=exportlist foo.o
```
  - Manually create the export file. If necessary, in a text editor, edit the export file to control which symbols will be exported when you create the shared library.
3. Create the shared library from the desired object files, using the **-qmkshrobj** compiler option and the **-bE** linker option if you created an export file in step 2. If you do not specify a **-bE** option, all symbols will be exported. (If you are creating a shared library from C++ object files, you can also assign an initialization priority to the shared library, as described in “Assigning priorities to objects” on page 46.) For example:

```
xlc -qmkshrobj foo.o -o mySharedObject -bE:exportlist
```

(The default name of the shared object is **shr.o**, unless you use the **-o** option to specify another name.)

Alternatively, if you are creating a shared library from C++ object files you can use the **makeC++SharedLib** utility, described in “Creating a shared library with the makeC++SharedLib utility” on page 56; however, the **-qmkshrobj** method is preferred as it has several advantages, including the ability to automatically handle C++ template instantiation, and compatibility with the **-O5** optimization option.

- Optionally, use the AIX **ar** command to produce an archive library file from multiple shared or static objects. For example:  

```
ar -rv libfoo.a shr.o anotherlibrary.so
```
- Link the shared library to the main application, as described in “Linking a library to an application” on page 45.

To create a shared library that uses run-time linking:

- Follow steps 1 and 2 in the procedure described above.
- Use the **-G** option to create a shared library from the generated object files, to be linked at load-time, and the **-bE** linker option to specify the name of the export list file. (You can also use the **-qmkshrobj** option if you want to specify a priority for a C++ shared object; see “Initializing static objects in libraries (C++)” on page 45.) For example:  

```
xlc -G -o libfoo.so foo1.o foo2.o -bE:exportlist
```
- Link the shared library to the main application, as described in “Linking a library to an application” on page 45.

**C++** If you want the system to perform static initialization when dynamically loading a shared library, use the load and unload functions described in “Dynamically loading a shared library” on page 49.

## Exporting symbols with the CreateExportList utility

**CreateExportList** is a shell script that creates a file containing a list of all the global symbols found in a given set of object files. Note that this command is run automatically when you use the **-qmkshrobj** option, unless you specify an alternative export file with the **-qexpfile** command.

The syntax of the **CreateExportList** command is as follows:

```
▶▶ CreateExportList exp_list [-r] [-f file_list] [obj_files] [-X32] [-X64]
```

You can specify one or more of the following options:

- r** If specified, template prefixes are pruned. The resource file symbol (**\_\_rsrc**) is not added to the resource list.
- exp\_list* The name of a file that will contain a list of global symbols found in the object files. This file is overwritten each time the **CreateExportList** command is run.
- f file\_list** The name of a file that contains a list of object file names.
- obj\_files* One or more names of object files.
- X32** Generates names from 32-bit object files in the input list specified by **-f file\_list** or *obj\_files*. This is the default.

**-X64** Generates names from 64-bit object files in the input list specified by **-f** *file\_list* or *obj\_files*.

## Linking a library to an application

You can use the same command string to link a static or shared library to your main program. For example:

```
xlc -o myprogram main.c -Ldirectory -lfoo
```

where *directory* is the path to the directory containing the library.

If your library uses run-time linking, add the **-brtl** option to the command:

```
xlc -brtl -o myprogram main.c -Ldirectory -lfoo
```

By using the **-l** option, you instruct the linker to search in the directory specified via the **-L** option for `libfoo.so`; if it is not found, the linker searches for `libfoo.a`. For additional linkage options, including options that modify the default behavior, see the AIX **ld** documentation.

## Linking a shared library to another shared library

Just as you link modules into an application, you can create dependencies between shared libraries by linking them together. For example:

```
xlc -qmkshrobj [-G] -o mylib.so myfile.o -Ldirectory -lfoo
```

## Related references

- **-qmkshrobj** in *XL C/C++ Compiler Reference*
- **-l** in *XL C/C++ Compiler Reference*
- **-L** in *XL C/C++ Compiler Reference*
- **ar** in the *AIX Commands Reference*
- **ld** in the *AIX Commands Reference*
- **-G** in *XL C/C++ Compiler Reference*
- **-brtl** in *XL C/C++ Compiler Reference*
- **-qexpfile** in *XL C/C++ Compiler Reference*

---

## Initializing static objects in libraries (C++)

The C++ language definition specifies that, before the **main** function in a C++ program is executed, all objects with constructors, from all the files included in the program must be properly constructed. Although the language definition specifies the order of initialization for these objects *within* a file (which follows the order in which they are declared), it does not, however, specify the order of initialization for these objects *across* files and libraries. You might want to specify the initialization order of static objects declared in various files and libraries in your program.

To specify an initialization order for objects, you assign relative *priority* numbers to objects. The mechanisms by which you can specify priorities for entire files or objects within files are discussed in “Assigning priorities to objects” on page 46. The mechanisms by which you can control the initialization order of objects across modules are discussed in “Order of object initialization across libraries” on page 48.

## Assigning priorities to objects

You can assign a priority number to objects and files within a single library, and the objects will be initialized at run time according to the order of priority. However, because of the differences in the way modules are loaded and objects initialized on the different platforms, the levels at which you can assign priorities vary among the different platforms, as follows:

▶ AIX ▶ Linux **Set the priority level for an entire file**

To use this approach, you specify the **-qpriority** compiler option during compilation. By default, all objects within a single file are assigned the same priority level, and are initialized in the order in which they are declared, and terminated in reverse declaration order.

▶ AIX ▶ Linux ▶ Mac OS X **Set the priority level for objects within a file**

To use this approach, you include **#pragma priority** directives in the source files. Each **#pragma priority** directive sets the priority level for all objects that follow it, until another pragma directive is specified. Within a file, the first **#pragma priority** directive must have a higher priority number than the number specified in the **-qpriority** option (if it is used), and subsequent **#pragma priority** directives must have increasing numbers. While the relative priority of objects *within* a single file will remain the order in which they are declared, the pragma directives will affect the order in which objects are initialized *across* files. The objects are initialized according to their priority, and terminated in reverse priority order.

▶ Linux ▶ Mac OS X **Set the priority level for individual objects**

To use this approach, you use **init\_priority** variable attributes in the source files. The **init\_priority** attribute takes precedence over **#pragma priority** directives, and can be applied to objects in any declaration order. On Linux, the objects are initialized according to their priority and terminated in reverse priority *across* compilation units; on Mac OS X, the objects are initialized according to their priority and terminated in reverse priority only *within* a compilation unit.

▶ AIX On AIX only, you can additionally set the priority of an entire shared library, by using the priority sub-option of the **-qmkshrobj** compiler option. As loading and initialization on AIX occur as separate processes, priority numbers assigned to files (or to objects within files) are entirely independent of priority numbers assigned to libraries, and do not need to follow any sequence.

## Using priority numbers

▶ AIX Priority numbers can range from -2147483643 to 2147483647. However, numbers from -2147483648 to -2147482624 are reserved for system use. The smallest priority number that you can specify, -2147482623, is initialized first. The largest priority number, 2147483647, is initialized last. If you do not specify a priority level, the default priority is 0 (zero).

▶ Linux ▶ Mac OS X Priority numbers can range from 101 to 65535. The smallest priority number that you can specify, 101, is initialized first. The largest priority number, 65535, is initialized last. If you do not specify a priority level, the default priority is 65535.

The examples below show how to specify the priority of objects within a single file, and across two files. “Order of object initialization across libraries” on page 48 provides detailed information on the order of initialization of objects on the AIX platform.



## Example of object initialization within a file

The following example shows how to specify the priority for several objects within a source file.

```
...
#pragma priority(2000) //Following objects constructed with priority 2000
...

static Base a ;

House b ;
...
#pragma priority(3000) //Following objects constructed with priority 3000
...

Barn c ;
...
#pragma priority(2500) // Error - priority number must be larger
                        // than preceding number (3000)
...
#pragma priority(4000) //Following objects constructed with priority 4000
...

Garage d ;
...
```

## Example of object initialization across multiple files

The following example describes the initialization order for objects in two files, farm.C and zoo.C. Both files use **#pragma priority** directives and are compiled with the **-qpriority** option.

```
farm.C -qpriority=2000                zoo.C -qpriority=2000
#pragma priority(3000)                ...
...                                    Lion k ;
Dog a ;                                #pragma priority(4000)
Dog b ;                                Bear m ;
...                                    ...
#pragma priority(6000)                #pragma priority(5000)
...                                    ...
Cat c ;                                Zebra n ;
Cow d ;                                Snake s ;
...                                    ...
#pragma priority(7000)                #pragma priority(8000)
Mouse e ;                               Frog f ;
...                                    ...
```

At run time, the objects in these files are initialized in the following order:

| Sequence | Object  | Priority value | Comment   |
|----------|---------|----------------|---|
| 1        | Lion k  | 2000           | Takes priority number of file zoo.o (2000) (initialized first). |
| 2        | Dog a   | 3000           | Takes pragma priority (3000).                                   |
| 3        | Dog b   | 3000           | Follows Dog a.  |
| 4        | Bear m  | 4000           | Next priority number, specified by pragma (4000).               |
| 5        | Zebra n | 5000           | Next priority number from pragma (5000).                        |
| 6        | Snake s | 5000           | Follows with same priority.                                     |
| 7        | Cat c   | 6000           | Next priority number.   |
| 8        | Cow d   | 6000           | Follows with same priority.                                     |

| Sequence | Object  | Priority value | Comment                                  |
|----------|---------|----------------|--|
| 9        | Mouse e | 7000           | Next priority number.                    |
| 10       | Frog f  | 8000           | Next priority number (initialized last). |

### Related references

- `-qmkshrobj` in *XL C/C++ Compiler Reference*
- `-qpriority` in *XL C/C++ Compiler Reference*
- `#pragma priority` in *XL C/C++ Compiler Reference*

## Order of object initialization across libraries

At run time, once all modules in an application have been loaded, the modules are initialized in their order of priority (the executable program containing the **main** function is always assigned a priority of 0). When objects are initialized within a library, the order of initialization follows the rules outlined in “Assigning priorities to objects” on page 46. If objects do not have priorities assigned, or have the same priorities, object files are initialized in random order, and the objects within the files are initialized according to their declaration order. Objects are terminated in reverse order of their construction.

### Example of object initialization across libraries

In this example, the following modules are used:

- `main.out`, the executable containing the main function
- `libS1` and `libS2`, two shared libraries
- `libS3` and `libS4`, two shared libraries that are dependencies of `libS1`
- `libS5` and `libS6`, two shared libraries that are dependencies of `libS2`

The dependent libraries are created with the following command strings:

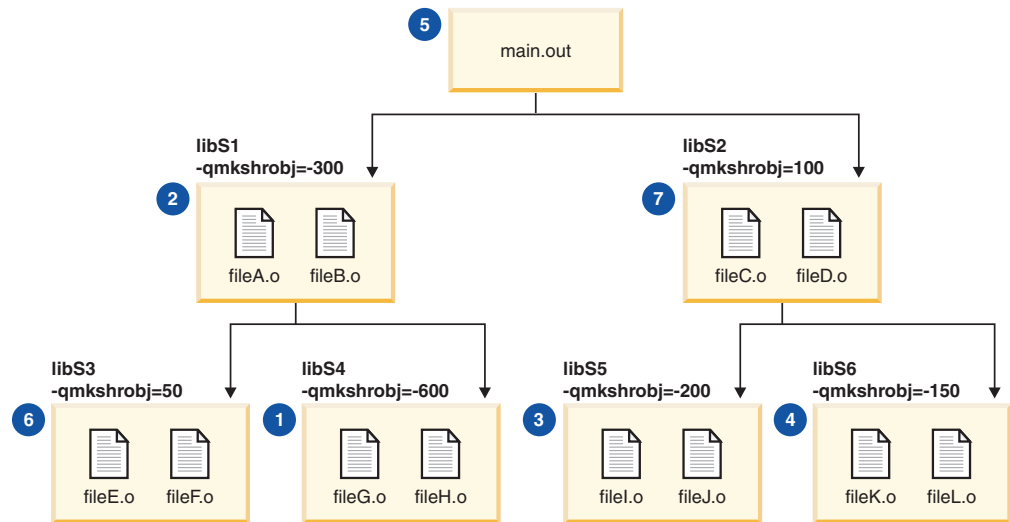
```
xlc -qmkshrobj=50 -o libS3 fileE.o fileF.o
xlc -qmkshrobj=-600 -o libS4 fileG.o fileH.o
xlc -qmkshrobj=-200 -o libS4 fileI.o fileJ.o
xlc -qmkshrobj=-150 -o libS6 fileK.o fileL.o
```

The parent libraries are linked with the main program with the following command strings:

```
xlc -qmkshrobj=-300 main.c -o main.out -L. -lA
xlc -qmkshrobj=100 main.c -o main.out -L. -lB
```

The following diagram shows the initialization order of the objects in the shared libraries.

Figure 1. Object initialization order on AIX



Objects are initialized as follows:

| Sequence | Object  | Priority value | Comment   |
|----------|---|----------------|---|
| 1        | libS4   | -600           | Initialized first (lowest priority number).   |
| 2        | libS1   | -300           | Initialized next (next priority number).  |
| 3        | libS5   | -200           | Initialized next (next priority number).  |
| 4        | libS6   | -150           | Initialized next (next priority number).  |
| 5        | main.out  | 0              | Initialized next (next priority number). The main program always has a priority of 0. |
| 6        | libS3   | 50             | Initialized next (next priority number).  |
| 7        | libS2   | 100            | Initialized last (next priority number).  |
| 8        | All objects from all libraries are initialized according to their priority numbers. |                |   |

## Related references

- `-qmksprobj` in *XL C/C++ Compiler Reference*

## Dynamically loading a shared library

If you want to programmatically control the loading and initialization of C++ objects contained in shared libraries, you can use two functions provided by XL C/C++: `loadAndInit` and `terminateAndUnload`. These functions are declared in the header file `load.h`, and you can call them from the main program to load, initialize, terminate, and unload any named shared library. These functions work in the same way as the AIX `load` and `unload` routines, but they additionally perform initialization of C++ objects.

**Note:** For portability, you might wish to use the POSIX `dlopen` and `dlclose` functions, which also perform initialization and termination, and interact

correctly with **loadAndInit** and **terminateAndUnload**. For more information on **dlopen** and **dlclose**, see the *AIX Technical Reference: Base Operating System and Extensions*.

## Loading and initializing a module with the **loadAndInit** function

The **loadAndInit** function takes the same parameters and returns the same values and error codes as the **load** routine. See the **load** routine in the *AIX Technical Reference: Base Operating System and Extensions* for more information.

### Format

```
#include <load.h>
int (*loadAndInit(char *FilePath, unsigned int Flags, char *LibraryPath))();
```

### Description

The **loadAndInit** function calls the AIX **load** routine to load the specified module (shared library) into the calling process's address space. If the shared library is loaded successfully, any C++ initialization is performed. The **loadAndInit** function ensures that a shared library is only initialized once, even if **dlopen** is used to load the library too. Subsequent loads of the same shared library will not perform any initialization of the shared library.

If loading a shared library results in other shared libraries being loaded, the initialization for those shared libraries will also be performed (if it has not been previously). If loading a shared library results in the initialization of multiple shared libraries, the order of initialization is determined by the priority assigned to the shared libraries when they were built. Shared libraries with the same priority are initialized in random order.

To terminate and unload the shared library, use the **terminateAndUnload** function, described below.

Do not reference symbols in the C++ initialization that need to be resolved by a call to the AIX **loadbind** routine, since the **loadbind** routine normally is not called until after the **loadAndInit** function returns.

### Parameters

#### *FilePath*

Points to the name of the shared library being loaded, or to the member of an archive. If you specify a relative or full path name (that is, a name containing one or more / characters), the file is used directly, and no search of directories specified in the *LibraryPath* is performed. If you specify a base name (that is, a name containing no / characters), a search is performed of the directory you specify in the *LibraryPath* parameter (see below).

*Flags*    Modifies the behavior of **loadAndInit**. If no special behavior is required, set the value to 0 (or 1). The possible flags are:

#### **L\_LIBPATH\_EXEC**

Specifies that the library path used at program execution time be prepended to any library path specified in the **loadAndInit** call. You should use this flag.

#### **L\_NOAUTODEFER**

Specifies that any deferred imports must be explicitly resolved by the use of the **loadbind** routine.

## L\_LOADMEMBER

Specifies that the *FilePath* is the name of a member in an archive. The format is *archivename.a(member)*.

### *LibraryPath*

Points to the default library search path.

## Return values

Upon successful completion, the **loadAndInit** function returns the pointer to function for the entry point (or data section) of the shared library.

If the **loadAndInit** function fails, a null pointer is returned, the module is not loaded or initialized, and the **errno** global variable is set to indicate the error.

## Terminating and unloading a module with the **terminateAndUnload** function

The **terminateAndUnload** function takes the same parameters and returns the same values and error codes as the **unload** routine. See the **unload** routine in *AIX Technical Reference: Base Operating System and Extensions* for more information.

## Format

```
#include <load.h>
int terminateAndUnload(int (*FunctionPointer)());
```

## Description

The **terminateAndUnload** function performs any C++ termination that is required and unloads the module (shared library). The function pointer returned by the **loadAndInit** routine is used as the parameter for the **terminateAndUnload** function. If this is the last time the shared library is being unloaded, any C++ termination is performed for this shared library and any other shared libraries that are being unloaded for the last time as well. The **terminateAndUnload** function ensures that the shared library is only terminated once, even if **dlclose** is used to unload the library too. The order of termination is the reverse order of initialization performed by the **loadAndInit** function. If any uncaught exceptions occur during the C++ termination, the termination is stopped and the shared library is unloaded.

If the **loadAndInit** function is called more times for a shared library than **terminateAndUnload**, the shared library will never have the C++ termination performed. If you rely on the C++ termination being performed at the time the **terminateAndUnload** function is called, ensure the number of calls to the **terminateAndUnload** function matches the number of calls to the **loadAndInit** function. If any shared libraries loaded with the **loadAndInit** function are still in use when the program exits, the C++ termination is performed.

If the **terminateAndUnload** function is used to unload shared libraries not loaded with the **loadAndInit** function, no termination will be performed.

## Parameters

### *FunctionPointer*

Specifies the name of the function returned by the **loadAndInit** function.

## Return values

Successful completion of the **terminateAndUnload** function returns a value of 0, even if the C++ termination was not performed and the shared library was not unloaded because the shared library was still in use.

If the **terminateAndUnload** function fails, it returns a value of -1 and sets **errno** to indicate the error.

---

## Chapter 8. Using the C++ utilities

XL C/C++ Enterprise Edition for AIX ships with a set of additional utilities you can use for managing your C++ applications:

- A filter for demangling compiled symbol names in object files. Described in “Demangling compiled C++ names with `c++filt`.”
- A library of classes for demangling and manipulating mangled names. Described in “Demangling compiled C++ names with the demangle class library” on page 54.
- A distributable shell script for creating shared libraries from library files. Described in “Creating a shared library with the `makeC++SharedLib` utility” on page 56.
- A distributable shell script for linking C++ object files and archives. Described in “Linking with the `linkxlC` utility” on page 57.

---

### Demangling compiled C++ names

When XL C/C++ compiles a C++ program, it encodes (mangles) all function names and certain other identifiers to include type and scoping information. The name mangling is necessary to accommodate overloading of C++ functions and operators. The linker uses these mangled names to resolve duplicate symbols and ensure type-safe linkage. These mangled names appear in the object files and final executable file.

Tools that can manipulate the files, the AIX **dump** utility for example, have only the mangled names and not the original source-code names, and present the mangled name in their output. This output might be undesirable because the names are no longer recognizable.

Two utilities convert the mangled names to their original source code names:

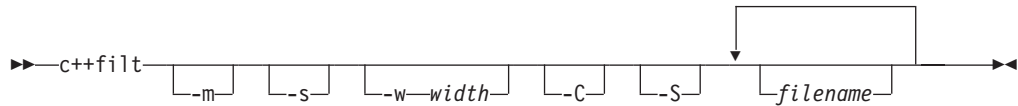
|                   |  |
|-------------------|--|
| <b>c++filt</b>    | A filter that demangles (decodes) mangled names.                               |
| <b>demangle.h</b> | A class library that you can use to develop tools to manipulate mangled names. |

Both are described in the following sections.

### Demangling compiled C++ names with `c++filt`

The `c++filt` utility is a filter that copies characters from file names or standard input to standard output, replacing all mangled names with their corresponding demangled names. You can use the filter directly with file name arguments, and the filter will output the demangled names of all mangled names in the files; or you can use a shell command that inputs text, such as specific mangled names, and pipe it to the filter, so that the filter provides the demangled names of the names you specified.

The syntax of the `c++filt` utility is as follows:



You can specify one or more of the following options:

- m** Produces a symbol map, containing a side-by-side listing of demangled names in the left column and their corresponding mangled names in the right column.
- s** Produces a continuous listing of each demangled name followed immediately by its mangled name.
- w *width*** Prints demangled names in fields *width* characters wide. If the name is shorter than *width*, it is padded on the right with blanks; if longer, it is truncated to *width*.
- C** Demangles standalone class names, such as Q2\_1X1Y.
- S** Demangles special compiler-generated symbol names, such as `__vft1X` (represents a virtual function).
- filename*** Is the name of the file containing the mangled names you want to demangle. You can specify more than one file name.

For example, the following command would show the symbols contained in an object file `functions.o`, producing a side-by-side listing of the mangled and demangled names with a field width of 40 characters:

```
c++filt -m -w 40 functions.o
```

The output would appear as follows:

| C++ Symbol Mapping           | Mangled:                |
|------------------------------|-------------------------|
| demangled:                   |                         |
| Average::insertValue(double) | insertValue__7AverageFd |
| Average::getCount()          | getCount__7AverageFv    |
| Average::getTotal()          | getTotal__7AverageFv    |
| Average::getAverage()        | getAverage__7AverageFv  |

The following command would show the demangled name immediately followed by the mangled name:

```
echo getAverage__7AverageFv | c++filt -s
```

The output would appear as follows:

```
Average::getAverage()getAverage__7AverageFv
```

## Demangling compiled C++ names with the demangle class library

The **demangle** class library contains a small class hierarchy that client programs can use to demangle names and examine the resulting parts of the name. It also provides a C-language interface for use in C programs. Although it is a C++ library, it uses no external C++ features, so you can link it directly to C programs. The **demangle** library is included as part of **libC.a**, and is automatically linked, when required, if **libC.a** is linked.

The header file declares a base class, **Name**, and a member function, **Demangle**, that takes a mangled name as a parameter, and returns the corresponding



demangled name. The header file declares four additional subclasses, which each contain member functions that allow you to get additional information about the name. These classes are:

**ClassName**

Can be used to query names of independent or nested classes.

**FunctionName**

Can be used to query names of functions.

**MemberVarName**

Can be used to query names of member variables.

**MemberFunctionName**

Can be used to query names of member functions.

For each of these classes, functions are defined that allow you to get information about the name. For example, for function names, a set of functions are defined that return the following information:

**Kind** Returns the type of the name being queried (that is, class, function, member variable, or member function).

**Text** Returns the fully qualified original text of the function.

**Rootname**

Returns the unqualified original name of the function.

**Arguments**

Returns the original text of the parameter list.

**Scope** Returns the original text of the function's qualifiers.

**IsConst/IsVolatile/IsStatic**

Returns true/false for these type qualifiers or storage class specifiers.

To demangle a name (represented as a character array), create a dynamic instance of the **Name** class, providing the character string to the class's constructor. For example, if the compiler mangled `X::f(int)` to the mangled name `f__1XFi`, in order to demangle the name, use the following code:

```
char *rest;
Name *name = Demangle("f__1XFi", rest) ;
```

If the supplied character string is not a name that requires demangling, because the original name was not mangled, the **Demangle** function returns `NULL`.

Once your program has constructed an instance of class **Name**, the program could query the instance to find out what kind of name it is, using the **Kind** method. Using the example of the mangled name `f__1XFi`, the following code:

```
name->Kind()
```

would return `MemberFunction`.

Based on the kind of name returned, the program could ask for the text of the different parts of the name, or the text of the entire name. The following table shows examples, still assuming the mangled name `f__1XFi`.

| To return...                         | ...use this code:  | Result |
|--------------------------------------|--|--------|
| The name of the function's qualifier | <code>((MemberFunctionName *)name)-&gt;Scope()-&gt;Text()</code> | X      |

| To return...                             | ...use this code:                        | Result    |
|--|--|-----------|
| The unqualified name of the function     | ((MemberFunctionName *)name)->RootName() | f         |
| The fully qualified name of the function | ((MemberFunctionName *)name)->Text()     | X::f(int) |

For further details about the **demangle** library and the C++ interface, look at the comments in the library's header file, */usr/vacpp/include/demangle.h*.

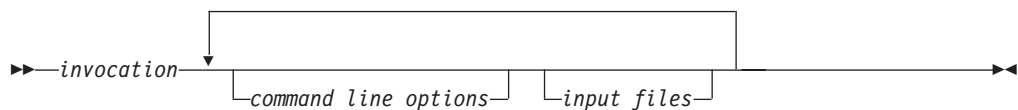
---

## Creating a shared library with the **makeC++SharedLib** utility

**makeC++SharedLib** is a shell script that links C++ **.o** and **.a** files. It can be redistributed and used by someone who does not have XL C/C++ installed.

It is recommended that you use the **-qmkshrobj** compiler option instead of the **makeC++SharedLib** command. Among the advantages to using this option are the automatic handling of link-time C++ template instantiation (using either the template include directory or the template registry), and compatibility with the **-O5** option.

The syntax for **makeC++SharedLib** is as follows:



*invocation* Is the command, preceded by the path. The following commands are provided:

- **makeC++SharedLib**
- **makeC++SharedLib\_r**
- **makeC++SharedLib\_r7**
- **makeC++SharedLib128**

### Command line options

- o *shared\_file.o*** The name of the file that will hold the shared file information. The default is **shr.o**.
- b** Uses the **-b** binder options of the **ld** command.
- L *lib\_dir*** Uses the **-L** option of the **ld** command to add the directory *lib\_dir* to the list of directories to be searched for unresolved symbols.
- l *library*** Adds *library* to the list of libraries to be searched for unresolved symbols.
- p *priority*** Specifies the priority level for the file. *priority* can be any number from -214782623 (highest priority-initialized first) to 214783647 (lowest priority-initialized last). Numbers from -214783648 to -214782624 are reserved for system use. For more information, see "Assigning priorities to objects" on page 46.
- I *import\_list*** Uses the **-bI** option of the **ld** command to resolve the list of symbols in the file *import\_list* that can be resolved by the binder.
- E *export\_list*** Uses the **-bE** option of the **ld** command to export the external

symbols in the *export\_list* file. If you do not specify **-E export\_list**, a list of all global symbols is generated.

- e file** Saves in *file* the list computed by **-E export\_list**.
- n name** Sets the entry name for the shared executable to *name*. This is equivalent to using the command **ld -e name**.
- X mode** Specifies the type of object file **makeC++SharedLib** should create. The mode must be either **32**, which processes only 32-bit object files, or **64**, which processes only 64-bit object files. The default is to process 32-bit object files (ignore 64-bit objects). You can also set the mode with the **OBJECT\_MODE** environment variable. For example, **OBJECT\_MODE=64** causes **makeC++SharedLib** to process any 64-bit objects and ignore 32-bit objects. The **-X** flag overrides the **OBJECT\_MODE** variable.

### Input Files

- file.o* Is an object file to be put into the shared library.
- file.a* Is an archive file to be put into the shared library.

---

## Linking with the linkx1C utility

**linkx1C** is a small shell script that links C++ **.o** and **.a** files. It can be redistributed and used by someone who does not have XL C/C++ installed.

**linkx1C** supports the following subset of the x1C compiler options:

- **-q32** (build a 32-bit application)
- **-q64** (build a 64-bit application)
- **-b** (pass linker options to **ld**)
- **-f** (pass a list of object files to **ld**)
- **-l** (pass a library to **ld**)
- **-L** (pass a library path to **ld**)
- **-o** (specify the output file)
- **-s** (strip output)
- **-qtwolink** (enable two-step linking)

**linkx1C** does not support the following compiler options:

- **-G**
- **-p**
- **-pg**

**linkx1C** accepts and ignores all other compiler options.

Unlike x1C, **linkx1C** does not specify any run-time libraries. You must specify these libraries yourself. For example, x1C **a.o** would become:

```
linkx1C a.o -L/usr/lpp/vacpp/lib -lC -lm -lc
```

## Related references

- **ld** in the *AIX Commands Reference*



---

## Chapter 9. Optimizing your applications

By default, a standard compilation performs only very basic local optimizations on your code, while still providing fast compilation and full debugging support. Once you have developed, tested, and debugged your code, you will want to take advantage of the extensive range of optimization capabilities offered by XL C/C++, that allow for significant performance gains without the need for any manual re-coding effort. In fact, it is not recommended to excessively hand-optimize your code (for example, by manually unrolling loops), as unusual constructs can confuse the compiler, and make your application difficult to optimize for new machines.

Instead, you can control XL C/C++ compiler optimization through the use of a set of compiler options. These options provide you with the following approaches to optimizing your code:

- You can use an option that performs a specific type of optimization, including:
  - System architecture. If your application will run on a specific hardware configuration, the compiler can generate instructions that are optimized for the target machine, including microprocessor architecture, cache or memory geometry, and addressing model. These options are discussed in “Optimizing for system architecture” on page 63.
  - Shared memory parallelization. If your application will run on hardware that supports shared memory parallelization, you can instruct the compiler to automatically generate threaded code, or to recognize OpenMP standard programming constructs. Options for parallelizing your program are discussed in “Using shared-memory parallelism” on page 65.
  - High-order loop analysis and transformation. The compiler uses various techniques to optimize loops. These options are discussed in “Using high-order loop analysis and transformations” on page 64.
  - Interprocedural analysis (IPA). The compiler reorganizes code sections to optimize calls between functions. IPA options are discussed in “Using interprocedural analysis” on page 66.
  - Profile-directed feedback (PDF). The compiler can optimize sections of your code based on call and block counts and execution times. PDF options are discussed in “Using profile-directed feedback” on page 67
  - Other types of optimization, including loop unrolling, function inlining, stack storage compacting, and many others. Brief descriptions of these options are provided in “Other optimization options” on page 70.
- You can use an optimization *level*, which bundles several techniques and may include one or more of the aforementioned specific optimization options. There are four optimization levels that perform increasingly aggressive optimizations on your code. Optimization levels are described in “Using optimization levels” on page 60.
- You can combine optimization options and levels to achieve the precise results you want. Discussions on how to do so are provided throughout the sections referenced above.

Keep in mind that program optimization implies a trade-off, in that it results in longer compile times, increased program size and disk usage, and diminished debugging capability. At higher levels of optimization, program semantics might be affected, and code that executed correctly before optimization might no longer run as expected. Thus, not all optimizations are beneficial for all applications or even

all portions of applications. For programs that are not computationally intensive, the benefits of faster instruction sequences brought about by optimization can be outweighed by better paging and cache performance brought about by a smaller program footprint.

To identify modules of your code that would benefit from performance enhancements, compile the selected files with the **-p** or **-pg** options, and use the operating system profiler **gprof** to identify functions that are "hot spots" and are computationally intensive. If both size and speed are important, optimize the modules which contain hot spots, while keeping code size compact in other modules. To find the right balance, you might need to experiment with different combinations of techniques.

An exhaustive list of all options available for optimization, organized by category, is provided in "Summary of options for optimization and performance" on page 71.

Finally, if you want to manually tune your application to complement the optimization techniques used by the compiler, Chapter 10, "Coding your application to improve performance," on page 73 provides suggestions and best practices for coding for performance.

---

## Related references

- **-p** in *XL C/C++ Compiler Reference*
- **-pg** in *XL C/C++ Compiler Reference*

---

## Using optimization levels

By default, the compiler performs only quick local optimizations such as constant folding and elimination of local common sub-expressions, while still allowing full debugging support. You can optimize your program by specifying various optimization levels, which provide increasing application performance, at the expense of larger program size and debugging support. The options you can specify are summarized in the following table, and more detailed descriptions of the techniques used at each optimization level are provided below.

Table 13. Optimization levels

| Option  | Behavior   |
|---|--|
| <b>-O</b> or <b>-O2</b> or <b>-qoptimize</b> or <b>-qoptimize=2</b> | Comprehensive low-level optimization; partial debugging support.           |
| <b>-O3</b> or <b>-qoptimize=3</b>                                   | More extensive optimization; some precision trade-offs.                    |
| <b>-O4</b> or <b>-qoptimize=4</b>                                   | Interprocedural optimization; loop optimization; automatic machine tuning. |
| <b>-O5</b> or <b>-qoptimize=5</b>                                   |  |

## Techniques used in optimization level 2

At optimization level 2, the compiler is conservative in the optimization techniques it applies and should not affect program correctness. At optimization level 2, the following techniques are used:

- Eliminating common sub-expressions that are recalculated in subsequent expressions. For example, with these expressions:

```
a = c + d;  
f = c + d + e;
```

the common expression  $c + d$  is saved from its first evaluation and is used in the subsequent statement to determine the value of  $f$ .

- Simplifying algebraic expressions. For example, the compiler combines multiple constants that are used in the same expression.
- Evaluating constants at compile time.
- Eliminating unused or redundant code, including:
  - Code that cannot be reached.
  - Code whose results are not subsequently used.
  - Store instructions whose values are not subsequently used.
- Rearranging the program code to minimize branching logic, combine physically separate blocks of code, and minimize execution time.
- Allocating variables and expressions to available hardware registers using a graph coloring algorithm.
- Replacing less efficient instructions with more efficient ones. For example, in array subscripting, an add instruction replaces a multiply instruction.
- Moving invariant code out of a loop, including:
  - Expressions whose values do not change within the loop.
  - Branching code based on a variable whose value does not change within the loop.
  - Store instructions.
- Unrolling some loops (equivalent to using the **-qunroll** compiler option).
- Pipelining some loops

### Techniques used in optimization level 3

At optimization levels 3 and above, the compiler is more aggressive, making changes to program semantics that will improve performance even if there is some risk that these changes will produce different results. Here are some examples:

- In some cases,  $X*Y*Z$  will be calculated as  $X*(Y*Z)$  instead of  $(X*Y)*Z$ . This could produce a different result due to rounding.
- In some cases, the sign of a negative zero value will be lost. This could produce a different result if you multiply the value by infinity.

“Getting the most out of optimization levels 2 and 3” on page 62 provides some suggestions for mitigating this risk.

At optimization level 3, all of the techniques in optimization level 2 are used, plus the following:

- Unrolling deeper loops and improving loop scheduling.
- Increasing the scope of optimization.
- Performing optimizations with marginal or niche effectiveness, which might not help all programs.
- Performing optimizations that are expensive in compile time or space.
- Reordering some floating-point computations, which might produce precision differences or affect the generation of floating-point-related exceptions (equivalent to compiling with the **-qnostrict** option).
- Eliminating implicit memory usage limits (equivalent to compiling with the **-qmaxmem=-1** option).

- Increasing automatic inlining.
- Propagating constants and values through structure copies.
- Removing the "address taken" attribute if possible after other optimizations.
- Grouping loads, stores and other operations on contiguous aggregate members, in some cases using VMX vector register operations.



## Techniques used in optimization levels 4 and 5

At optimization levels 4 and 5, all of the techniques in optimization levels 2 and 3 are used, plus the following:

- Interprocedural analysis, which invokes the optimizer at link time to perform optimizations across multiple source files (equivalent to compiling with the **-qipa** option).
- High-order transformations, which provide optimized handling of loop nests and array language constructs (equivalent to compiling with the **-qhot** option).
- Hardware-specific optimization (equivalent to compiling with the **-qarch=auto**, **-qtune=auto**, and **-qcache=auto** options).
- At optimization level 5, more detailed interprocedural analysis (the equivalent to compiling with the **-qipa=level=2** option). With level 2 IPA, high-order transformations (equivalent to compiling with **-qhot**) are delayed until link time, after whole-program information has been collected.

## Getting the most out of optimization levels 2 and 3

Here is a recommended approach to using optimization levels 2 and 3:

1. If possible, test and debug your code without optimization before using **-O2**.
2. Ensure that your code complies with its language standard.
3.  In C code, ensure that the use of pointers follows the type restrictions: generic pointers should be **char\*** or **void\***. Also check that all shared variables and pointers to shared variables are marked **volatile**.
4.  In C, use the **-qlibansi** compiler option unless your program defines its own functions with the same names as library functions.
5. Compile as much of your code as possible with **-O2**.
6. If you encounter problems with **-O2**, consider using **-qalias=noansi** rather than turning off optimization.
7. Next, use **-O3** on as much code as possible.
8. If you encounter problems or performance degradations, consider using **-qstrict** or **-qcompact** along with **-O3** where necessary.
9. If you still have problems with **-O3**, switch to **-O2** for a subset of files, but consider using **-qmaxmem=-1**, **-qnostrict**, or both.

## Related references

- **-O** in *XL C/C++ Compiler Reference*
- **-qnostrict** in *XL C/C++ Compiler Reference*
- **-qmaxmem** in *XL C/C++ Compiler Reference*
- **-qunroll** in *XL C/C++ Compiler Reference*
- **-qalias** in *XL C/C++ Compiler Reference*
- **-qlibansi** in *XL C/C++ Compiler Reference*



---

## Optimizing for system architecture

You can instruct the compiler to generate code for optimal execution on a given microprocessor or architecture family. By selecting appropriate target machine options, you can optimize to suit the broadest possible selection of target processors, a range of processors within a given family of processor architectures, or a specific processor. The following table lists the optimization options that affect individual aspects of the target machine. Using a predefined optimization level sets default values for these individual options.

Table 14. Target machine options

| Option         | Behavior  |
|----------------|---|
| <b>-q32</b>    | Generates code for a 32-bit (4/4/4) addressing model (32-bit execution mode). This is the default setting.  |
| <b>-q64</b>    | Generates code for a 64-bit (4/8/8) addressing model (64-bit execution mode).   |
| <b>-qarch</b>  | Selects a family of processor architectures for which instruction code should be generated. This option restricts the instruction set generated to a subset of that for the PowerPC <sup>®</sup> architecture. The default is <b>-qarch=com</b> . Using <b>-O4</b> or <b>-O5</b> sets the default to <b>-qarch=auto</b> . |
| <b>-qtune</b>  | Biases optimization toward execution on a given microprocessor, without implying anything about the instruction set architecture to use as a target. The default is <b>-qtune=pwr3</b> .  |
| <b>-qcache</b> | Defines a specific cache or memory geometry. The defaults are determined through the setting of <b>-qtune</b> .   |

For a complete listing of valid hardware-related suboptions and combinations of suboptions, see “Specify Compiler Options for Architecture-Specific, 32- or 64-bit Compilation”, and “Acceptable Compiler Mode and Processor Architecture Combinations” in *XL C/C++ Compiler Reference*.

## Getting the most out of target machine options

### Using -qarch options

If your application will run on the same machine on which you are compiling it, you can use the **-qarch=auto** option, which automatically detects the specific architecture of the compiling machine, and generates code to take advantage of instructions available only on that machine (or on a system that supports the equivalent processor architecture). Otherwise, try to specify with **-qarch** the smallest family of machines possible that will be expected to run your code reasonably well.

To optimize square root operations, by generating inline code rather than calling a library function, you need to specify a family of processors that supports `sqrt` functionality, in addition to specifying the **-qignerrno** option (or any optimization option that implies it). Use **-qarch=ppc64grsq**, which will generate correct code for all processors in the **ppc64grsq** group of processors: RS64 II, RS64 III, POWER3, POWER4, POWER5, and PowerPC970.

### Using -qtune options

If you specify a particular architecture with **-qarch**, **-qtune** will automatically select the suboption that generates instruction sequences with the best performance for that architecture. If you specify a *group* of architectures with **-qarch**, compiling with **-qtune=auto** will generate 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.

Try to specify with **-qtune** the particular architecture that the compiler should target for best performance but still allow execution of the produced object file on all architectures specified in the **-qarch** option. For information on the valid combinations of **-qarch** and **-qtune**, see “Acceptable Compiler Mode and Processor Architecture Combinations” in *XL C/C++ Compiler Reference*.

### Using **-qcache** options

Before using the **-qcache** option, use the **-qlistopt** option to generate a listing of the current settings and verify if they are satisfactory. If you decide to specify your own **-qcache** suboptions, use **-qhot** or **-qsmp** along with it. For the full set of suboptions, option syntax, and guidelines for use, see **-qcache** in *XL C/C++ Compiler Reference*.

### Related references

- **-qarch** in *XL C/C++ Compiler Reference*
- **-qcache** in *XL C/C++ Compiler Reference*
- **-qtune** in *XL C/C++ Compiler Reference*
- **-qlistopt** in *XL C/C++ Compiler Reference*

---

## Using high-order loop analysis and transformations

High-order transformations are optimizations that specifically improve the performance of loops through techniques such as interchange, fusion, and unrolling. The goals of these loop optimizations include:

- Reducing the costs of memory access through the effective use of caches and translation look-aside buffers.
- Overlapping computation and memory access through effective utilization of the data prefetching capabilities provided by the hardware.
- Improving the utilization of microprocessor resources through reordering and balancing the usage of instructions with complementary resource requirements.

To enable high-order loop analysis and transformations, you use the **-qhot** option. The following table lists the suboptions available for **-qhot**.

Table 15. **-qhot** suboptions

| suboption | Behavior   |
|-----------|--|
| vector    | Instructs the compiler to transform some loops to use optimized versions of various trigonometric functions and operations such as reciprocal and square root that reside in a built-in library, rather than use the standard versions. The optimized versions make different trade-offs with respect to precision versus performance. This suboption is enabled by default when you use <b>-qhot</b> , <b>-O4</b> , or <b>-O5</b> . |
| novector  | Instructs the compiler to avoid optimizations that use the above-mentioned built-in library functions. Use this suboption or <b>-qstrict</b> if you do not want your precision of your program’s results to be affected.   |
| arraypad  | Instructs the compiler to pad any arrays where it infers there might be a benefit and to pad by whatever amount it chooses.  |

## Getting the most out of -qhot

Here are some suggestions for using **-qhot**:

- Try using **-qhot** along with **-O2** and **-O3** for all of your code. It is designed to have a neutral effect when no opportunities for transformation exist.
- If you encounter unacceptably long compile times (this can happen with complex loop nests) or if your performance degrades with the use of **-qhot**, try using **-qhot=novector**, or **-qstrict** or **-qcompact** along with **-qhot**.
- If necessary, deactivate **-qhot** selectively, allowing it to improve some of your code.

## Related references

- **-qhot** in *XL C/C++ Compiler Reference*
- **-qstrict** in *XL C/C++ Compiler Reference*

---

## Using shared-memory parallelism

Some IBM pSeries™ machines are capable of shared-memory parallel processing. You can compile with **-qsmp** to generate the threaded code needed to exploit this capability. The option implies an optimization level of at least **-O2**.

The following table lists the most commonly used suboptions. Descriptions and syntax of all the suboptions are provided in *XL C/C++ Compiler Reference*.

Table 16. Commonly used **-qsmp** suboptions

| suboption          | Behavior  |
|--------------------|---|
| auto               | Instructs the compiler to automatically generate parallel code where possible without user assistance. This is the default setting if you do not specify any <b>-qsmp</b> suboptions, and it also implies the <b>opt</b> suboption. |
| omp                | Instructs the compiler to observe OpenMP language extensions for specifying explicit parallelism. Note that <b>-qsmp=omp</b> is currently incompatible with <b>-qsmp=auto</b> .   |
| opt                | Instructs the compiler to optimize as well as parallelize. The optimization is equivalent to <b>-O2 -qhot</b> in the absence of other optimization options.   |
| <i>fine_tuning</i> | Other values for the suboption provide control over thread scheduling, nested parallelism, locking, etc.  |

## Getting the most out of -qsmp

Here are some suggestions for using the **-qsmp** option:

- Before using **-qsmp** with automatic parallelization, test your programs using optimization and **-qhot** in a single-threaded manner.
- If you are compiling an OpenMP program and do not want automatic parallelization, use **-qsmp=omp:noauto**.
- Always use the reentrant compiler invocations (the *\_r* invocations) when using **-qsmp**.
- By default, the run-time environment uses all available processors. Do not set the **XLSMPOPTS=PARHDS** or **OMP\_NUM\_THREADS** environment variables unless you want to use fewer than the number of available processors. You might want to set the number of executing threads to a small number or to 1 to ease debugging.
- If you are using a dedicated machine or node, consider setting the **SPINS** and **YIELDS** environment variables (suboptions of the **XLSMPOPTS** environment

variables) to 0. Doing so prevents the operating system from intervening in the scheduling of threads across synchronization boundaries such as barriers.

- When debugging an OpenMP program, try using **-qsmp=noopt** (without **-O**) to make the debugging information produced by the compiler more precise.

## Related references

- **-qsmp** in *XL C/C++ Compiler Reference*
- "Runtime Options for Parallel Processing" in *XL C/C++ Compiler Reference*
- "OpenMP Runtime Options for Parallel Processing" in *XL C/C++ Compiler Reference*

---

## Using interprocedural analysis

Interprocedural analysis (IPA) enables the compiler to optimize across different files (whole-program analysis), and can result in significant performance improvements. You can specify interprocedural analysis on the compile step only or on both compile and link steps ("whole program" mode). Whole program mode expands the scope of optimization to an entire program unit, which can be an executable or shared object. As IPA can significantly increase compilation time, you should limit using IPA to the final performance tuning stage of development.

You enable IPA by specifying the **-qipa** option. The most commonly used suboptions and their effects are described in the following table. The full set of suboptions and syntax is described in **-qipa** in the *XL C/C++ Compiler Reference*.

Table 17. Commonly used **-qipa** suboptions

| suboption               | Behavior  |
|-------------------------|---|
| level=0                 | Program partitioning and simple interprocedural optimization, which consists of: <ul style="list-style-type: none"> <li>• Automatic recognition of standard libraries.</li> <li>• Localization of statically bound variables and procedures.</li> <li>• Partitioning and layout of procedures according to their calling relationships. (Procedures that call each other frequently are located closer together in memory.)</li> <li>• Expansion of scope for some optimizations, notably register allocation.</li> </ul>   |
| level=1                 | Inlining and global data mapping. Specifically: <ul style="list-style-type: none"> <li>• Procedure inlining.</li> <li>• Partitioning and layout of static data according to reference affinity. (Data that is frequently referenced together will be located closer together in memory.)</li> </ul> <p>This is the default level if you do not specify any suboptions with the <b>-qipa</b> option.</p>   |
| level=2                 | Global alias analysis, specialization, interprocedural data flow: <ul style="list-style-type: none"> <li>• Whole-program alias analysis. This level includes the disambiguation of pointer dereferences and indirect function calls, and the refinement of information about the side effects of a function call.</li> <li>• Intensive intraprocedural optimizations. This can take the form of value numbering, code propagation and simplification, code motion into conditions or out of loops, elimination of redundancy.</li> <li>• Interprocedural constant propagation, dead code elimination, pointer analysis, and code motion across functions.</li> <li>• Procedure specialization (cloning).</li> </ul> |
| inline= <i>variable</i> | Allows you precise control over function inlining.  |

Table 17. Commonly used **-qipa** suboptions (continued)

| suboption          | Behavior  |
|--------------------|---|
| <i>fine_tuning</i> | Other values for <b>-qipa</b> provide the ability to specify the behavior of library code, tune program partitioning, read commands from a file, etc. |

## Getting the most from **-qipa**

It is not necessary to compile everything with **-qipa**, but try to apply it to as much of your program as possible. Here are some suggestions:

- Specify the **-qipa** option on both the compile and link steps of the entire application, or as much of it as possible. Although you can also use **-qipa** with libraries, shared objects, and executables, be sure to use **-qipa** to compile the main and exported functions.
- When compiling and linking separately, use **-qipa=noobject** on the compile step for faster compilation.
- When specifying optimization options in a makefile, remember to use the compiler driver (**xlc**) to link, and to include all compiler options on the link step.
- As IPA can generate significantly larger object files than traditional compilations, ensure that there is enough space in the **/tmp** directory (at least 200 MB), or use the **TMPDIR** environment variable to specify a different directory with sufficient free space.
- Try varying the **level** suboption if link time is too long. Compiling with **-qipa=level=0** can still be very beneficial for little additional link time.
- Use **-qlist** or **-qipa=list** to generate a report of functions that were inlined. If too few or too many functions are inlined, consider using **-qipa=inline** or **-qipa=noinline**. To control inlining of specific functions, use **-Q+** or **-Q-**.

## Related references

- **-qipa** in *XL C/C++ Compiler Reference*
- **-Q** in *XL C/C++ Compiler Reference*
- **-qlist** in *XL C/C++ Compiler Reference*

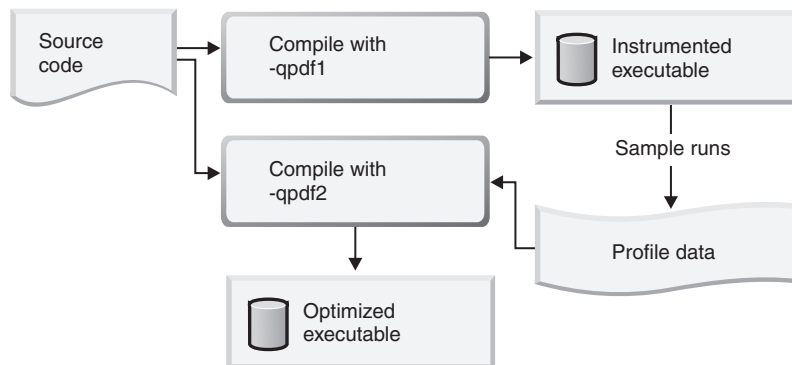
---

## Using profile-directed feedback

You can use profile-directed feedback (PDF) to tune the performance of your application for a typical usage scenario. The compiler optimizes the application based on an analysis of how often branches are taken and blocks of code are executed. Because the process requires compiling the entire application twice, it is intended to be used after other debugging and tuning is finished, as one of the last steps before putting the application into production.

The following diagram illustrates the PDF process.

Figure 2. Profile-directed feedback



You first compile the program with the **-qpdf1** option, which generates profile data by using the compiled program in the same ways that users will typically use it. You then compile the program again, with the **-qpdf2** option. This optimizes the program based on the profile data, by invoking **qipa=level=0**.

Note that you do not need to compile all of the application's code with the **-qpdf1** option to benefit from the PDF process; in a large application, you might want to concentrate on those areas of the code that can benefit most from optimization.

To use the **-qpdf** options:

1. Compile some or all of the source files in the application with **-qpdf1**.
2. Run the application using a typical data set or several typical data sets. It is important to use data that is representative of the data that will be used by your application in a real-world scenario. When the application exits, it writes profiling information to the PDF file in the current working directory or the directory specified by the **PDFDIR** environment variable.
3. Compile the application with **-qpdf2**.

You can take more control of the PDF file generation, as follows:

1. Compile some or all of the source files in the application with **-qpdf1**.
2. Run the application using a typical data set or several typical data sets. This produces a PDF file in the current directory.
3. Change the directory specified by the **PDFDIR** environment variable to produce a PDF file in a different directory.
4. Re-compile the application with **-qpdf1**.
5. Repeat steps 3 and 4 as often as you want.
6. Use the **mergepdf** utility to combine the PDF files into one PDF file. For example, if you produce three PDF files that represent usage patterns that will occur 53%, 32%, and 15% of the time respectively, you can use this command:  

```
mergepdf -r 53 path1 -r 32 path2 -r 15 path3
```
7. Compile the application with **-qpdf2**.

To collect more detailed information on function call and block statistics, do the following:

1. Compile the application with **-qpdf1 -qshowpdf**.

2. Run the application using a typical data set or several typical data sets. The application writes more detailed profiling information in the PDF file.
3. Use the **showpdf** utility to view the information in the PDF file.

To erase the information in the PDF directory, use the **cleanpdf** utility or the **resetpdf** utility.

## Example of compilation with pdf and showpdf

The following example shows how you can use PDF with the **showpdf** utility to view the call and block statistics for a “Hello World” application.

The source for the program file `hello.c` is as follows:

```
#include <stdio.h>
void HelloWorld()
{
    printf("Hello World");
}
main()
{
    HelloWorld();
    return 0;
}
```

1. Compile the source file:
 

```
xlc -qpdf1 -qshowpdf hello.c
```
2. Run the resulting program executable **a.out**.
3. Run the **showpdf** utility to display the call and block counts for the executable:
 

```
showpdf
```

The results will look similar to the following:

```
HelloWorld(4): 1 (hello.c)

Call Counters:
5 | 1 printf(6)

Call coverage = 100% ( 1/1 )

Block Counters:
3-5 | 1
6 |
6 | 1

Block coverage = 100% ( 2/2 )

-----
main(5): 1 (hello.c)

Call Counters:
10 | 1 HelloWorld(4)

Call coverage = 100% ( 1/1 )

Block Counters:
8-11 | 1
11 |

Block coverage = 100% ( 1/1 )

Total Call coverage = 100% ( 2/2 )
Total Block coverage = 100% ( 3/3 )
```

## Related references


- `-qpdf` in *XL C/C++ Compiler Reference*
- `-showpdf` in *XL C/C++ Compiler Reference*

---

## Other optimization options

The following options are available to control particular aspects of optimization. They are often enabled as a group or given default values when you enable a more general optimization option or level. For more information on these options, see the heading for each option in the *XL C/C++ Compiler Reference*.

Table 18. Selected compiler options for optimizing performance

| Option  | Description   |
|---|---|
| <code>-qcompact</code>  | Suppresses optimizations that would result in larger code size, such as loop unrolling, and function inlining.  |
| <code>-qignerrno</code>   | Allows the compiler to assume that <code>errno</code> is not modified by library function calls, so that such calls can be optimized. Also allows optimization of square root operations, by generating inline code rather than calling a library function. (For processors that support <code>sqrt</code> .) |
| <code>-qsmallstack</code>   | Instructs the compiler to compact stack storage. Doing so may increase heap usage.  |
| <code>-qinline</code>   | Controls inlining of named functions. Can be used at compile time, link time, or both. When <code>-qipa</code> is used, <code>-qinline</code> is synonymous with <code>-qipa=inline</code> .  |
| <code>-qunroll</code>   | Independently controls loop unrolling. Is implicitly activated under <code>-O3</code> .   |
| <code>-qinlglue</code>  | Instructs the compiler to inline the "glue code" generated by the linker and used to make a call to an external function or a call made through a function pointer. 64-bit mode only.   |
| <code>-qtbtable</code>  | Controls the generation of traceback table information. 64-bit mode only.   |
|  <code>-qnoeh</code> | Informs the compiler that no C++ exceptions will be thrown and that cleanup code can be omitted. If your program does not throw any C++ exceptions, use this option to compact your program by removing exception-handling code.  |
| <code>-qnounwind</code>   | Informs the compiler that the stack will not be unwound while any routine in this compilation is active. This option can improve optimization of non-volatile register saves and restores. In C++, the <code>-qnounwind</code> option implies the <code>-qnoeh</code> option.                                 |
| <code>-qnostrict</code>   | Stops the compiler from reordering floating-point calculations and potentially excepting instructions. A potentially excepting instruction is one that might raise an interrupt due to erroneous execution (for example, floating-point overflow, a memory access violation).                                 |
| <code>-qlargepage</code>  | Supports large 16M pages in addition to the default 4K pages, to allow hardware prefetching to be done more efficiently. Informs the compiler that heap and static data will be allocated from large pages at execution time.   |



## Summary of options for optimization and performance

The following table presents a summary of the compiler options that deal with optimization and performance tuning. The options are grouped by type. For a description, full syntax, and usage of each option, see the appropriate option heading in the *XL C/C++ Compiler Reference*.

Table 19. Options related to optimization and performance tuning

| Optimization flags                                   | Optimization restriction options                                  |
|--|---|
| -O/-qoptimize<br>-qagrrcopy                          | -qkeepparm<br>-qnoprefetch<br>-qstrict<br>-qcompact<br>-qmaxmem   |
| Function inlining                                    | Code size reduction   |
| -Q<br>-qinline<br>-qinlglue                          | -bmaxdata<br>-s<br>-qnoeh   |
| Side effects   | Loop optimization   |
| -qignerrno<br>-qisolated_call                        | -qhot<br>-qreport<br>-qnostrict_induction<br>-qunroll             |
| Whole-program analysis                               | Processor and architectural optimization                          |
| -qipa  | -qarch<br>-qcache<br>-qtune<br>-qdirectstorage                    |
| Performance data collection                          | Other optimization options  |
| -qfdpr<br>-p<br>-qpdf1<br>-qpdf2<br>-pg<br>-qshowpdf | -qlargepage<br>-qtocdata<br>-qtocmerge<br>-qsmallstack<br>-qspill |



---

## Chapter 10. Coding your application to improve performance

Chapter 9, “Optimizing your applications,” on page 59 discusses the various compiler options that XL C/C++ provides for optimizing your code with minimal coding effort. If you want to take your application a step further, to complement and take the most advantage of compiler optimizations, the following sections discuss C and C++ programming techniques that can improve performance of your code:

- “Find faster input/output techniques”
- “Reduce function-call overhead”
- “Manage memory efficiently” on page 75
- “Optimize variables” on page 75
- “Manipulate strings efficiently” on page 76
- “Optimize expressions and program logic” on page 77
- “Optimize operations in 64-bit mode” on page 77

---

### Find faster input/output techniques

There are a number of ways to improve your program’s performance of input and output:





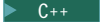

- Use binary streams instead of text streams. In binary streams, data is not changed on input or output.
- Use the low-level I/O functions, such as **open** and **close**. These functions are faster and more specific to the application than the stream I/O functions like **fopen** and **fclose**. You must provide your own buffering for the low-level functions.
- If you do your own I/O buffering, make the buffer a multiple of 4K, which is the size of a page.
- When reading input, read in a whole line at once rather than one character at a time.
- If you know you have to process an entire file, determine the size of the data to be read in, allocate a single buffer to read it to, read the whole file into that buffer at once using **read**, and then process the data in the buffer. This reduces disk I/O, provided the file is not so big that excessive swapping will occur. Consider using the **mmap** function to access the file.
- Instead of **scanf** and **fscanf**, use **fgets** to read in a string, and then use one of **atoi**, **atol**, **atof**, or **\_atold** to convert it to the appropriate format.
- Use **sprintf** only for complicated formatting. For simpler formatting, such as string concatenation, use a more specific string function.


---

### Reduce function-call overhead

When you write a function or call a library function, consider the following guidelines:

- Call a function directly, rather than using function pointers.
- Pass a value to a function as an argument, rather than letting the function take the value from a global variable.

- Use constant arguments in inlined functions whenever possible. Functions with constant arguments provide more opportunities for optimization.
- Use the `#pragma isolated_call` preprocessor directive to list functions that have no side effects and do not depend on side effects.
- Use `#pragma disjoint` within functions for pointers or reference parameters that can never point to the same memory.
- Declare a nonmember function as static whenever possible. This can speed up calls to the function.
-  Usually, you should not declare virtual functions inline. If all virtual functions in a class are inline, the virtual function table and all the virtual function bodies will be replicated in each compilation unit that uses the class.
-  When declaring functions, use the `const` specifier whenever possible.
-  Fully prototype all functions. A full prototype gives the compiler and optimizer complete information about the types of the parameters. As a result, promotions from unwidened types to widened types are not required, and parameters can be passed in appropriate registers.
-  Avoid using unprototyped variable argument functions.
- Design functions so that the most frequently used parameters are in the leftmost positions in the function prototype.
- Avoid passing by value structures or unions as function parameters or returning a structure or a union. Passing such aggregates requires the compiler to copy and store many values. This is worse in C++ programs in which class objects are passed by value because a constructor and destructor are called when the function is called. Instead, pass or return a pointer to the structure or union, or pass it by reference.
- Pass non-aggregate types such as `int` and `short` by value rather than passing by reference, whenever possible.
- If your function exits by returning the value of another function with the same parameters that were passed to your function, put the parameters in the same order in the function prototypes. The compiler can then branch directly to the other function.
- Use the built-in functions, which include string manipulation, floating-point, and trigonometric functions, instead of coding your own. Intrinsic functions require less overhead and are faster than a function call, and often allow the compiler to perform better optimization.
  -  Your functions are automatically mapped to built-in functions if you include the XL C/C++ header files.
  -  Your functions are mapped to built-in functions if you include `math.h` and `string.h`.
- Selectively mark your functions for inlining, using the `inline` keyword. An inlined function requires less overhead and is generally faster than a function call. The best candidates for inlining are small functions that are called frequently from a few places, or functions called with one or more compile-time constant parameters, especially those that affect `if`, `switch` or `for` statements. You might also want to put these functions into header files, which allows automatic inlining across file boundaries even at low optimization levels. Be sure to inline all functions that only load or store a value, or use simple operators such as comparison or arithmetic operators. Large functions and functions that are called rarely might not be good candidates for inlining.

- Avoid breaking your program into too many small functions. If you must use small functions, seriously consider using the `-qipa` compiler option, which can automatically inline such functions, and uses other techniques for optimizing calls between functions.
-  Avoid virtual functions and virtual inheritance unless required for class extensibility. These language features are costly in object space and function invocation performance.





## Related references

- `#pragma isolated_call` in *XL C/C++ Compiler Reference*
- `#pragma disjoint` in *XL C/C++ Compiler Reference*
- `-qipa` in *XL C/C++ Compiler Reference*

---

## Manage memory efficiently

Because C++ objects are often allocated from the heap and have limited scope, memory use affects performance more in C++ programs than it does in C programs. For that reason, consider the following guidelines when you develop C++ applications:

- In a structure, declare the largest members first.
- In a structure, place variables near each other if they are frequently used together.
-  Ensure that objects that are no longer needed are freed or otherwise made available for reuse. One way to do this is to use an *object manager*. Each time you create an instance of an object, pass the pointer to that object to the object manager. The object manager maintains a list of these pointers. To access an object, you can call an object manager member function to return the information to you. The object manager can then manage memory usage and object reuse.
- Storage pools are a good way of keeping track of used memory (and reclaiming it) without having to resort to an object manager or reference counting.
-  Avoid copying large, complicated objects.
-  Avoid performing a *deep copy* if a *shallow copy* is all you require. For an object that contains pointers to other objects, a shallow copy copies only the pointers and not the objects to which they point. The result is two objects that point to the same contained object. A deep copy, however, copies the pointers and the objects they point to, as well as any pointers or objects contained within that object, and so on.
-  Use virtual methods only when absolutely necessary.

---

## Optimize variables

Consider the following guidelines:

- Use local variables, preferably automatic variables, as much as possible. The compiler must make several worst-case assumptions about a global variable. For example, if a function uses external variables and also calls external functions, the compiler assumes that every call to an external function could change the value of every external variable. If you know that a global variable is not affected by any function call, and this variable is read several times with function calls interspersed, copy the global variable to a local variable and then use this local variable.

- If you must use global variables, use static variables with file scope rather than external variables whenever possible. In a file with several related functions and static variables, the optimizer can gather and use more information about how the variables are affected.
- If you must use external variables, group external data into structures or arrays whenever it makes sense to do so. All elements of an external structure use the same base address.
- The `#pragma isolated_call` preprocessor directive can improve the run-time performance of optimized code by allowing the compiler to make less pessimistic assumptions about the storage of external and static variables. Isolated call functions with constant or loop-invariant parameters can be moved out of loops, and multiple calls with the same parameters can be replaced with a single call.
- Avoid taking the address of a variable. If you use a local variable as a temporary variable and must take its address, avoid reusing the temporary variable. Taking the address of a local variable inhibits optimizations that would otherwise be done on calculations involving that variable.
- Use constants instead of variables where possible. The optimizer will be able to do a better job reducing run-time calculations by doing them at compile-time instead. For instance, if a loop body has a constant number of iterations, use constants in the loop condition to improve optimization (for (i=0; i<4; i++) can be better optimized than for (i=0; i<x; i++)).
- Use register-sized integers (**long** data type) for scalars. For large arrays of integers, consider using one- or two-byte integers or bit fields.
- Use the smallest floating-point precision appropriate to your computation. Use the **long double** data type only when extremely high precision is required.

## Related references

- `#pragma isolated_call` in *XL C/C++ Compiler Reference*

---

## Manipulate strings efficiently

The handling of string operations can affect the performance of your program.

- When you store strings into allocated storage, align the start of the string on an 8-byte boundary.
- Keep track of the length of your strings. If you know the length of a string, you can use **mem** functions instead of **str** functions. For example, **memcpy** is faster than **strcpy** because it does not have to search for the end of the string.
- If you are certain that the source and target do not overlap, use **memcpy** instead of **memmove**. This is because **memcpy** copies directly from the source to the destination, while **memmove** might copy the source to a temporary location in memory before copying to the destination (depending on the length of the string).
- When manipulating strings using **mem** functions, faster code will be generated if the *count* parameter is a constant rather than a variable. This is especially true for small count values.
- Make string literals read-only, whenever possible. This improves certain optimization techniques and reduces memory usage if there are multiple uses of the same string. You can explicitly set strings to read-only by using **#pragma strings (readonly)** in your source files or **-qro** (this is enabled by default) to avoid changing your source files.

## Related references

- `#pragma strings (readonly)` in *XL C/C++ Compiler Reference*
- `-qro` in *XL C/C++ Compiler Reference*

---

## Optimize expressions and program logic

Consider the following guidelines:

- If components of an expression are used in other expressions, assign the duplicated values to a local variable.
- Avoid forcing the compiler to convert numbers between integer and floating-point internal representations. For example:


```
float array[10];
float x = 1.0;
int i;
for (i = 0; i < 9; i++) {      /* No conversions needed */
    array[i] = array[i]*x;
    x = x + 1.0;
}
for (i = 0; i < 9; i++) {      /* Multiple conversions needed */
    array[i] = array[i]*i;
}
```

When you must use mixed-mode arithmetic, code the integer and floating-point arithmetic in separate computations whenever possible.

- Avoid **goto** statements that jump into the middle of loops. Such statements inhibit certain optimizations.
- Improve the predictability of your code by making the fall-through path more probable. Code such as:  
`if (error) {handle error} else {real code}`

should be written as:

```
if (!error) {real code} else {error}
```

- If one or two cases of a **switch** statement are typically executed much more frequently than other cases, break out those cases by handling them separately before the **switch** statement.
-  Use **try** blocks for exception handling only when necessary because they can inhibit optimization.
- Keep array index expressions as simple as possible.

---

## Optimize operations in 64-bit mode

The ability to handle larger amounts of data directly in physical memory rather than relying on disk I/O 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.

Some ways to compensate for the performance liabilities of 64-bit programs include:

- Avoid performing mixed 32- and 64-bit operations. For example, adding a 32-bit data type to a 64-bit data type requires that the 32-bit type be sign-extended to clear the upper 32 bits of the register. This slows the computation.
- Avoid 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, and folded, at compile time only.

- Use **long** types instead of **signed**, **unsigned**, and plain **int** types for variables which will be frequently accessed, such as loop counters and array indexes. Doing so frees the compiler from having to truncate or sign-extend array references, parameters during function calls, and function results during returns.



---

## Appendix. Memory debug library functions

This appendix contains reference information about the XL C/C++ memory debug library functions, which are extensions of the standard C memory management functions. The appendix is divided into two sections:

- “Memory allocation debug functions” describes the debug versions of the standard library functions for allocating heap memory.
- “String handling debug functions” on page 87 describes the debug versions of the standard library functions for manipulating strings.

To use these debug versions, you can do either of the following:

- In your source code, prefix any of the default or user-defined-heap memory management functions with `_debug_`.
- If you do not wish to make changes to the source code, simply compile with the `-qheapdebug` option. This option maps all calls to memory management functions to their debug version counterparts. To prevent a call from being mapped, parenthesize the function name.

All of the examples provided in this appendix assume compilation with the `-qheapdebug` option.

---

### Related references

- `-qheapdebug` in *XL C/C++ Compiler Reference*
- 

## Memory allocation debug functions

This section describes the debug versions of standard and user-created heap memory allocation functions. All of these functions automatically make a call to `_heap_check` or `_uheap_check` to check the validity of the heap. You can then use the `_dump_allocated` or `_dump_allocated_delta` functions to print the information returned by the heap-checking functions.

### `_debug_calloc` — Allocate and initialize memory

#### Format

```
#include <stdlib.h> /* also in <malloc.h> */
void *_debug_calloc(size_t num, size_t size, const char *file, size_t line);
```

#### Description

This is the debug version of `calloc`. Like `calloc`, it allocates memory from the default heap for an array of `num` elements, each of length `size` bytes. It then initializes all bits of each element to 0. In addition, `_debug_calloc` makes an implicit call to `_heap_check`, and stores the name of the file `file` and the line number `line` where the storage is allocated.

#### Return value

Returns a pointer to the reserved space. If not enough memory is available, or if `num` or `size` is 0, returns NULL.

## Example

This example reserves storage of 100 bytes. It then attempts to write to storage that was not allocated. When `_debug_malloc` is called again, `_heap_check` detects the error, generates several messages, and stops the program.

```
/* _debug_malloc.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *ptr1, *ptr2;

    if (NULL == (ptr1 = (char*)calloc(1, 100))) {
        puts("Could not allocate memory block.");
        exit(EXIT_FAILURE);
    }
    memset(ptr1, 'a', 105);          /* overwrites storage that was not allocated */
    ptr2 = (char*)calloc(2, 20);     /* this call to calloc invokes _heap_check */
    puts("_debug_malloc did not detect that a memory block was overwritten.");
    return 0;
}
```

The output should be similar to:

```
End of allocated object 0x00073890 was overwritten at 0x000738f4.
The first eight bytes of the memory block (in hex) are: 6161616161616161.
This memory block was (re)allocated at line number 9 in _debug_malloc.c.
Heap state was valid at line 9 of _debug_malloc.c.
Memory error detected at line 14 of _debug_malloc.c.
```

## `_debug_free` — Free allocated memory

### Format

```
#include <stdlib.h> /* also in <malloc.h> */
void _debug_free(void *ptr, const char *file, size_t line);
```

### Description

This is the debug version of `free`. Like `free`, it frees the block of memory pointed to by `ptr`. `_debug_free` also sets each block of freed memory to `0xFB`, so you can easily locate instances where your program uses the data in freed memory. In addition, `_debug_free` makes an implicit call to the `_heap_check` function, and stores the file name `file` and the line number `line` where the memory is freed.

Because `_debug_free` always checks the type of heap from which the memory was allocated, you can use this function to free memory blocks allocated by the regular, heap-specific, or debug versions of the memory management functions. However, if the memory was not allocated by the memory management functions, or was previously freed, `_debug_free` generates an error message and the program ends.

### Return value

There is no return value.

## Example

This example reserves two blocks, one of 10 bytes and the other of 20 bytes. It then frees the first block and attempts to overwrite the freed storage. When `_debug_free` is called a second time, `_heap_check` detects the error, prints out several messages, and stops the program.

```

/* _debug_free.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *ptr1, *ptr2;
    if (NULL == (ptr1 = (char*)malloc(10)) || NULL == (ptr2 = (char*)malloc(20))) {
        puts("Could not allocate memory block.");
        exit(EXIT_FAILURE);
    }
    free(ptr1);
    memset(ptr1, 'a', 5);      /* overwrites storage that has been freed */
    free(ptr2);               /* this call to free invokes _heap_check */
    puts("_debug_free did not detect that a freed memory block was overwritten.");
    return 0;
}

```

The output should be similar to:

```

Free heap was overwritten at 0x00073890.
Heap state was valid at line 12 of _debug_free.c.
Memory error detected at line 14 of _debug_free.c.

```

## **`_debug_heapmin` — Free unused memory in the default heap**

### **Format**

```

#include <stdlib.h> /* also in <malloc.h> */
int _debug_heapmin(const char *file, size_t line);

```

### **Description**

This is the debug version of `_heapmin`. Like `_heapmin`, it returns all unused memory from the default runtime heap to the operating system. In addition, `_debug_heapmin` makes an implicit call to `_heap_check`, and stores the file name *file* and the line number *line* where the memory is returned.

### **Return value**

If successful, returns 0; otherwise, returns -1.

### **Example**

This example allocates 10000 bytes of storage, changes the storage size to 10 bytes, and then uses `_debug_heapmin` to return the unused memory to the operating system. The program then attempts to overwrite memory that was not allocated. When `_debug_heapmin` is called again, `_heap_check` detects the error, generates several messages, and stops the program.

```

/* _debug_heapmin.c */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *ptr;

    /* Allocate a large object from the system */
    if (NULL == (ptr = (char*)malloc(10000))) {
        puts("Could not allocate memory block.");
        exit(EXIT_FAILURE);
    }
    ptr = (char*)realloc(ptr, 10);
    _debug_heapmin(); /* No allocation problems to detect */

    *(ptr - 1) = 'a'; /* Overwrite memory that was not allocated */
}

```

```

_heapmin();                /* This call to _heapmin invokes _heap_check */

puts("_debug_heapmin did not detect that a non-allocated memory block"
     "was overwritten.");
return 0;
}

```

Possible output is:

```

Header information of object 0x000738b0 was overwritten at 0x000738ac.
The first eight bytes of the memory block (in hex) are: AAAAAAAAAAAAAAAA.
This memory block was (re)allocated at line number 13 in _debug_heapmin.c.
Heap state was valid at line 14 of _debug_heapmin.c.
Memory error detected at line 17 of _debug_heapmin.c.

```

## **`_debug_malloc` — Allocate memory**

### **Format**

```

#include <stdlib.h> /* also in <malloc.h> */
void *_debug_malloc(size_t size, const char *file, size_t line);

```

### **Description**

This is the debug version of **malloc**. Like **malloc**, it reserves a block of storage of *size* bytes from the default heap. **\_debug\_malloc** also sets all the memory it allocates to 0xAA, so you can easily locate instances where your program uses the data in the memory without initializing it first. In addition, **\_debug\_malloc** makes an implicit call to **\_heap\_check**, and stores the file name *file* and the line number *line* where the storage is allocated.

### **Return value**

Returns a pointer to the reserved space. If not enough memory is available or if *size* is 0, returns NULL.

### **Example**

This example allocates 100 bytes of storage. It then attempts to write to storage that was not allocated. When **\_debug\_malloc** is called again, **\_heap\_check** detects the error, generates several messages, and stops the program.

```

/* _debug_malloc.c */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *ptr1, *ptr2;

    if (NULL == (ptr1 = (char*)malloc(100))) {
        puts("Could not allocate memory block.");
        exit(EXIT_FAILURE);
    }
    *(ptr1 - 1) = 'a'; /* overwrites storage that was not allocated */
    ptr2 = (char*)malloc(10); /* this call to malloc invokes _heap_check */
    puts("_debug_malloc did not detect that a memory block was overwritten.");
    return 0;
}

```

Possible output is:

```

Header information of object 0x00073890 was overwritten at 0x0007388c.
The first eight bytes of the memory block (in hex) are: AAAAAAAAAAAAAAAA.
This memory block was (re)allocated at line number 8 in _debug_malloc.c.
Heap state was valid at line 8 of _debug_malloc.c.
Memory error detected at line 13 of _debug_malloc.c.

```

## **`_debug_ucalloc` — Reserve and initialize memory from a user-created heap**

### **Format**

```
#include <umalloc.h>
void *_debug_ucalloc(Heap_t heap, size_t num, size_t size, const char *file, size_t line);
```

### **Description**

This is the debug version of `_ucalloc`. Like `_ucalloc`, it allocates memory from the *heap* you specify for an array of *num* elements, each of length *size* bytes. It then initializes all bits of each element to 0. In addition, `_debug_ucalloc` makes an implicit call to `_uheap_check`, and stores the name of the file *file* and the line number *line* where the storage is allocated.

If the *heap* does not have enough memory for the request, `_debug_ucalloc` calls the heap-expanding function that you specify when you create the heap with `_ucreate`.

**Note:** Passing `_debug_ucalloc` a heap that is not valid results in undefined behavior.

### **Return value**

Returns a pointer to the reserved space. If *size* or *num* was specified as zero, or if your heap-expanding function cannot provide enough memory, returns NULL.

### **Example**

This example creates a user heap and allocates memory from it with `_debug_ucalloc`. It then attempts to write to memory that was not allocated. When `_debug_free` is called, `_uheap_check` detects the error, generates several messages, and stops the program.

```
/* _debug_ucalloc.c */
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>
#include <string.h>

int main(void)
{
    Heap_t myheap;
    char *ptr;

    /* Use default heap as user heap */
    myheap = _udefault(NULL);

    if (NULL == (ptr = (char*)_ucalloc(myheap, 100, 1))) {
        puts("Cannot allocate memory from user heap.");
        exit(EXIT_FAILURE);
    }
    memset(ptr, 'x', 105); /* Overwrites storage that was not allocated */
    free(ptr);
    return 0;
}
```

The output should be similar to :

```
End of allocated object 0x00073890 was overwritten at 0x000738f4.
The first eight bytes of the memory block (in hex) are: 7878787878787878.
This memory block was (re)allocated at line number 14 in _debug_ucalloc.c.
Heap state was valid at line 14 of _debug_ucalloc.c.
Memory error detected at line 19 of _debug_ucalloc.c.
```

## **debug\_uheapmin — Free unused memory in a user-created heap**

### **Format**

```
#include <umalloc.h>
int _debug_uheapmin(Heap_t heap, const char *file, size_t line);
```

### **Description**

This is the debug version of `_uheapmin`. Like `_uheapmin`, it returns all unused memory blocks from the specified *heap* to the operating system.

To return the memory, `_debug_uheapmin` calls the heap-shrinking function you supply when you create the heap with `_ucreate`. If you do not supply a heap-shrinking function, `_debug_uheapmin` has no effect and returns 0.

In addition, `_debug_uheapmin` makes an implicit call to `_uheap_check` to validate the heap.

### **Return value**

If successful, returns 0. A nonzero return value indicates failure. If the heap specified is not valid, generates an error message with the file name and line number in which the call to `_debug_uheapmin` was made.

### **Example**

This example creates a heap and allocates memory from it, then uses `_debug_uheapmin` to release the memory.

```
/* _debug_uheapmin.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <umalloc.h>

int main(void)
{
    Heap_t myheap;
    char *ptr;

    /* Use default heap as user heap */
    myheap = _udefault(NULL);

    /* Allocate a large object */
    if (NULL == (ptr = (char*)_umalloc(myheap, 60000))) {
        puts("Cannot allocate memory from user heap.\n");
        exit(EXIT_FAILURE);
    }
    memset(ptr, 'x', 60000);
    free(ptr);

    /* _debug_uheapmin will attempt to return the freed object to the system */
    if (0 != _uheapmin(myheap)) {
        puts("_debug_uheapmin returns failed.\n");
        exit(EXIT_FAILURE);
    }
    return 0;
}
```

## **debug\_umalloc — Reserve memory blocks from a user-created heap**

### **Format**

```
#include <umalloc.h>
void *_debug_umalloc(Heap_t heap, size_t size, const char *file, size_t line);
```

## Description

This is the debug version of `_umalloc`. Like `_umalloc`, it reserves storage space from the *heap* you specify for a block of *size* bytes. `_debug_umalloc` also sets all the memory it allocates to `0xAA`, so you can easily locate instances where your program uses the data in the memory without initializing it first.

In addition, `_debug_umalloc` makes an implicit call to `_uheap_check`, and stores the name of the file *file* and the line number *line* where the storage is allocated.

If the *heap* does not have enough memory for the request, `_debug_umalloc` calls the heap-expanding function that you specify when you create the heap with `_ucreate`.

**Note:** Passing `_debug_umalloc` a heap that is not valid results in undefined behavior.

## Return value

Returns a pointer to the reserved space. If *size* was specified as zero, or your heap-expanding function cannot provide enough memory, returns `NULL`.

## Example

This example creates a heap `myheap` and uses `_debug_umalloc` to allocate 100 bytes from it. It then attempts to overwrite storage that was not allocated. The call to `_debug_free` invokes `_uheap_check`, which detects the error, generates messages, and ends the program.

```
/* _debug_umalloc.c */
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>
#include <string.h>

int main(void)
{
    Heap_t myheap;
    char *ptr;

    /* Use default heap as user heap */
    myheap = _udefault(NULL);

    if (NULL == (ptr = (char*)_umalloc(myheap, 100))) {
        puts("Cannot allocate memory from user heap.\n");
        exit(EXIT_FAILURE);
    }
    memset(ptr, 'x', 105); /* Overwrites storage that was not allocated */
    free(ptr);
    return 0;
}
```

The output should be similar to :

```
End of allocated object 0x00073890 was overwritten at 0x000738f4.
The first eight bytes of the memory block (in hex) are: 7878787878787878.
This memory block was (re)allocated at line number 14 in _debug_umalloc.c.
Heap state was valid at line 14 of _debug_umalloc.c.
Memory error detected at line 19 of _debug_umalloc.c.
```

## `_debug_realloc` — Reallocate memory block

### Format

```
#include <stdlib.h> /* also in <malloc.h> */
void *_debug_realloc(void *ptr, size_t size, const char *file, size_t line);
```

## Description

This is the debug version of `realloc`. Like `realloc`, it reallocates the block of memory pointed to by `ptr` to a new `size`, specified in bytes. It also sets any new memory it allocates to `0xAA`, so you can easily locate instances where your program tries to use the data in that memory without initializing it first. In addition, `_debug_realloc` makes an implicit call to `_heap_check`, and stores the file name `file` and the line number `line` where the storage is reallocated.

If `ptr` is `NULL`, `_debug_realloc` behaves like `_debug_malloc` (or `malloc`) and allocates the block of memory.

Because `_debug_realloc` always checks to determine the heap from which the memory was allocated, you can use `_debug_realloc` to reallocate memory blocks allocated by the regular or debug versions of the memory management functions. However, if the memory was not allocated by the memory management functions, or was previously freed, `_debug_realloc` generates an error message and the program ends.

## Return value

Returns a pointer to the reallocated memory block. The `ptr` argument is not the same as the return value; `_debug_realloc` always changes the memory location to help you locate references to the memory that were not freed before the memory was reallocated.

If `size` is 0, returns `NULL`. If not enough memory is available to expand the block to the given size, the original block is unchanged and `NULL` is returned.

## Example

This example uses `_debug_realloc` to allocate 100 bytes of storage. It then attempts to write to storage that was not allocated. When `_debug_realloc` is called again, `_heap_check` detects the error, generates several messages, and stops the program.

```
/* _debug_realloc.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *ptr;

    if (NULL == (ptr = (char*)realloc(NULL, 100))) {
        puts("Could not allocate memory block.");
        exit(EXIT_FAILURE);
    }
    memset(ptr, 'a', 105); /* overwrites storage that was not allocated */
    ptr = (char*)realloc(ptr, 200); /* realloc invokes _heap_check */
    puts("_debug_realloc did not detect that a memory block was overwritten." );
    return 0;
}
```

The output should be similar to:

```
End of allocated object 0x00073890 was overwritten at 0x000738f4.
The first eight bytes of the memory block (in hex) are: 6161616161616161.
This memory block was (re)allocated at line number 8 in _debug_realloc.c.
Heap state was valid at line 8 of _debug_realloc.c.
Memory error detected at line 13 of _debug_realloc.c.
```



## Related references

- “Functions for debugging memory heaps” on page 31

---

## String handling debug functions

This section describes the debug versions of the string manipulation and memory functions of the standard C string handling library. Note that these functions check only the current default heap; they do not check all heaps in applications that use multiple user-created heaps.

### **`_debug_memcpy` — Copy bytes**

#### **Format**

```
#include <string.h>
void *_debug_memcpy(void *dest, const void *src, size_t count, const char *file,
                    size_t line);
```

#### **Description**

This is the debug version of `memcpy`. Like `memcpy`, it copies `count` bytes of `src` to `dest`, where the behavior is undefined if copying takes place between objects that overlap.

`_debug_memcpy` validates the heap after copying the bytes to the target location, and performs this check only when the target is within a heap. `_debug_memcpy` makes an implicit call to `_heap_check`. If `_debug_memcpy` detects a corrupted heap when it makes a call to `_heap_check`, `_debug_memcpy` will report the file name `file` and line number `line` in a message.

#### **Return value**

Returns a pointer to `dest`.

#### **Example**

This example contains a programming error. On the call to `memcpy` used to initialize the target location, the count is more than the size of the target object, and the `memcpy` operation copies bytes past the end of the allocated object.

```
/* _debug_memcpy.c */
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define MAX_LEN 10

int main(void)
{
    char *source, *target;

    target = (char*)malloc(MAX_LEN);
    memcpy(target, "This is the target string", 11);

    printf("Target is \"%s\"\n", target);
    return 0;
}
```

The output should be similar to:

End of allocated object 0x00073c80 was overwritten at 0x00073c8a.  
The first eight bytes of the memory block (in hex) are: 5468697320697320.  
This memory block was (re)allocated at line number 11 in `_debug_memcpy.c`.  
Heap state was valid at line 11 of `_debug_memcpy.c`.  
Memory error detected at line 12 of `_debug_memcpy.c`.

## `_debug_memmove` — Copy bytes

### Format

```
#include <string.h>
void *_debug_memmove(void *dest, const void *src, size_t count, const char *file,
                    size_t line);
```

### Description

This is the debug version of `memmove`. Like `memmove`, it copies *count* bytes of *src* to *dest*, and allows for copying between objects that might overlap.

`_debug_memmove` validates the heap after copying the bytes to the target location, and performs this check only when the target is within a heap. `_debug_memmove` makes an implicit call to `_heap_check`. If `_debug_memmove` detects a corrupted heap when it makes a call to `_heap_check`, `_debug_memmove` will report the file name *file* and line number *line* in a message.

### Return value

Returns a pointer to *dest*.

### Example

This example contains a programming error. The count specified on the call to `memmove` is 15 instead of 5, and the `memmove` operation copies bytes past the end of the allocated object.

```
/* _debug_memmove.c */
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define SIZE      21

int main(void)
{
    char *target, *p, *source;

    target = (char*)malloc(SIZE);
    strcpy(target, "a shiny white sphere");
    p = target+8;                /* p points at the starting character
                                of the word we want to replace */
    source = target+2;          /* start of "shiny" */

    printf("Before memmove, target is \"%s\"\n", target);
    memmove(p, source, 15);
    printf("After memmove, target becomes \"%s\"\n", target);
    return 0;
}
```

The output should be similar to:

```
Before memmove, target is "a shiny white sphere"
End of allocated object 0x00073c80 was overwritten at 0x00073c95.
The first eight bytes of the memory block (in hex) are: 61207368696E7920.
This memory block was (re)allocated at line number 11 in _debug_memmove.c.
Heap state was valid at line 12 of _debug_memmove.c.
Memory error detected at line 18 of _debug_memmove.c.
```

## **`_debug_memset` — Set bytes to value**

### **Format**

```
#include <string.h>
void *_debug_memset(void *dest, int c, size_t count, const char *file, size_t line);
```

### **Description**

This is the debug version of `memset`. Like `memset`, it sets the first *count* bytes of *dest* to the value *c*. The value of *c* is converted to an unsigned character.

`_debug_memset` validates the heap after setting the bytes, and performs this check only when the target is within a heap. `_debug_memset` makes an implicit call to `_heap_check`. If `_debug_memset` detects a corrupted heap when it makes a call to `_heap_check`, `_debug_memset` will report the file name *file* and line number *line* in a message.

### **Return value**

Returns a pointer to *dest*.

### **Example**

This example contains a programming error. The invocation of `memset` that puts 'B' in the buffer specifies the wrong count, and stores bytes past the end of the buffer.

```
/* _debug_memset.c */
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define BUF_SIZE 20

int main(void)
{
    char *buffer, *buffer2;
    char *string;

    buffer = (char*)calloc(1, BUF_SIZE+1); /* +1 for null-terminator */

    string = (char*)memset(buffer, 'A', 10);
    printf("\nBuffer contents: %s\n", string);
    memset(buffer+10, 'B', 20);

    return 0;
}
```

The output should be:

```
Buffer contents: AAAAAAAAAA
End of allocated object 0x00073c80 was overwritten at 0x00073c95.
The first eight bytes of the memory block (in hex) are: 4141414141414141.
This memory block was (re)allocated at line number 12 in _debug_memset.c.
Heap state was valid at line 14 of _debug_memset.c.
Memory error detected at line 16 of _debug_memset.c.
```

## **`_debug_strcat` — Concatenate strings**

### **Format**

```
#include <string.h>
char *_debug_strcat(char *string1, const char *string2, const char *file, size_t file);
```

## Description

This is the debug version of `strcat`. Like `strcat`, it concatenates *string2* to *string1* and ends the resulting string with the null character.

`_debug_strcat` validates the heap after concatenating the strings, and performs this check only when the target is within a heap. `_debug_strcat` makes an implicit call to `_heap_check`. If `_debug_strcat` detects a corrupted heap when it makes a call to `_heap_check`, `_debug_strcat` will report the file name *file* and line number *file* in a message.

## Return value

Returns a pointer to the concatenated string *string1*.

## Example

This example contains a programming error. The `buffer1` object is not large enough to store the result after the string " program" is concatenated.

```
/* _debug_strcat.hc */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define SIZE 10

int main(void)
{
    char *buffer1;
    char *ptr;

    buffer1 = (char*)malloc(SIZE);
    strcpy(buffer1, "computer");

    ptr = strcat(buffer1, " program");
    printf("buffer1 = %s\n", buffer1);
    return 0;
}
```

The output should be similar to:

```
End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
The first eight bytes of the memory block (in hex) are: 636F6D7075746572.
This memory block was (re)allocated at line number 12 in _debug_strcat.c.
Heap state was valid at line 13 of _debug_strcat.c.
Memory error detected at line 15 of _debug_strcat.c.
```

## `_debug_strcpy` — Copy strings

### Format

```
#include <string.h>
char *_debug_strcpy(char *string1, const char *string2, const char *file, size_t line);
```

### Description

This is the debug version of `strcpy`. Like `strcpy`, it copies *string2*, including the ending null character, to the location specified by *string1*.

`_debug_strcpy` validates the heap after copying the string to the target location, and performs this check only when the target is within a heap. `_debug_strcpy` makes an implicit call to `_heap_check`. If `_debug_strcpy` detects a corrupted heap when it makes a call to `_heap_check`, `_debug_strcpy` will report the file name *file* and line number *line* in a message.

## Return value

Returns a pointer to the copied string *string1*.

## Example

This example contains a programming error. The source string is too long for the destination buffer, and the **strcpy** operation damages the heap.

```
/* _debug_strcpy.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define SIZE      10
int main(void)
{
    char *source = "1234567890123456789";
    char *destination;
    char *return_string;

    destination = (char*)malloc(SIZE);
    strcpy(destination, "abcdefg");

    printf("destination is originally = '%s'\n", destination);
    return_string = strcpy(destination, source);
    printf("After strcpy, destination becomes '%s'\n\n", destination);
    return 0;
}
```

The output should be similar to:

```
destination is originally = 'abcdefg'
End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
The first eight bytes of the memory block (in hex) are: 3132333435363738.
This memory block was (re)allocated at line number 13 in _debug_strcpy.c.
Heap state was valid at line 14 of _debug_strcpy.c.
Memory error detected at line 17 of _debug_strcpy.c.
```

## **\_\_debug\_strncat** — Concatenate strings

### Format

```
#include <string.h>
char *__debug_strncat(char *string1, const char *string2, size_t count,
                     const char *file, size_t line);
```

### Description

This is the debug version of **strncat**. Like **strncat**, it appends the first count characters of *string2* to *string1* and ends the resulting string with a null character (`\0`). If *count* is greater than the length of *string2*, the length of *string2* is used in place of *count*.

**\_\_debug\_strncat** validates the heap after appending the characters, and performs this check only when the target is within a heap. **\_\_debug\_strncat** makes an implicit call to **\_\_heap\_check**. If **\_\_debug\_strncat** detects a corrupted heap when it makes a call to **\_\_heap\_check**, **\_\_debug\_strncat** will report the file name *file* and line number *line* in a message.

### Return value

Returns a pointer to the joined string *string1*.

### Example

This example contains a programming error. The `buffer1` object is not large enough to store the result after eight characters from the string "programming" are concatenated.

```

/* _debug_strncat.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define SIZE      10
int main(void)
{
    char *buffer1;
    char *ptr;
    buffer1 = (char*)malloc(SIZE);
    strcpy(buffer1, "computer");
    /* Call strncat with buffer1 and " programming"          */
    ptr = strncat(buffer1, " programming", 8);
    printf("strncat: buffer1 = \"%s\"\n", buffer1);
    return 0;
}

```

The output should be similar to:

```

End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
The first eight bytes of the memory block (in hex) are: 636F6D7075746572.
This memory block was (re)allocated at line number 12 in _debug_strncat.c.
Heap state was valid at line 13 of _debug_strncat.c.
Memory error detected at line 17 of _debug_strncat.c.

```

## **\_debug\_strncpy — Copy strings**

### **Format**

```

#include <string.h>
char *_debug_strncpy(char *string1, const char *string2, size_t count,
                    const char *file, size_t line);

```

### **Description**

This is the debug version of **strncpy**. Like **strncpy**, it copies *count* characters of *string2* to *string1*. If *count* is less than or equal to the length of *string2*, a null character (`\0`) is not appended to the copied string. If *count* is greater than the length of *string2*, the *string1* result is padded with null characters (`\0`) up to length *count*.

**\_debug\_strncpy** validates the heap after copying the strings to the target location, and performs this check only when the target is within a heap. **\_debug\_strncpy** makes an implicit call to **\_heap\_check**. If **\_debug\_strncpy** detects a corrupted heap when it makes a call to **\_heap\_check**, **\_debug\_strncpy** will report the file name *file* and line number *line* in a message.

### **Return value**

Returns a pointer to *string1*.

### **Example**

This example contains a programming error. The source string is too long for the destination buffer, and the **strncpy** operation damages the heap.

```

/* _debug_strncpy */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define SIZE      10

```

```

int main(void)
{
    char *source = "1234567890123456789";
    char *destination;
    char *return_string;
    int index = 15;

    destination = (char*)malloc(SIZE);
    strcpy(destination, "abcdefg");

    printf("destination is originally = '%s'\n", destination);
    return_string = strncpy(destination, source, index);
    printf("After strncpy, destination becomes '%s'\n", destination);
    return 0;
}

```

The output should be similar to:

```

destination is originally = 'abcdefg'
End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
The first eight bytes of the memory block (in hex) are: 3132333435363738.
This memory block was (re)allocated at line number 14 in _debug_strncpy.c.
Heap state was valid at line 15 of _debug_strncpy.c.
Memory error detected at line 18 of _debug_strncpy.c.

```

## **`_debug_strnset` — Set characters in a string**

### **Format**

```

#include <string.h>
char *_debug_strnset(char *string, int c, size_t n, const char *file, size_t line);

```

### **Description**

This is the debug version of `strnset`. Like `strnset`, it sets, at most, the first  $n$  characters of `string` to `c` (converted to a `char`), where if  $n$  is greater than the length of `string`, the length of `string` is used in place of  $n$ .

`_debug_strnset` validates the heap after setting the bytes, and performs this check only when the target is within a heap. `_debug_strnset` makes an implicit call to `_heap_check`. If `_debug_strnset` detects a corrupted heap when it makes a call to `_heap_check`, `_debug_strnset` will report the file name `file` and line number `line` in a message.

### **Return value**

Returns a pointer to the altered `string`. There is no error return value.

### **Example**

This example contains two programming errors. The string, `str`, was created without a null-terminator to mark the end of the string, and without the terminator `strnset` with a count of 10 stores bytes past the end of the allocated object.

```

/* _debug_strnset */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(void)
{
    char *str;
    str = (char*)malloc(10);
    printf("This is the string after strnset: %s\n", str);
    return 0;
}

```

The output should be:

```
End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
The first eight bytes of the memory block (in hex) are: 7878787878797979.
This memory block was (re)allocated at line number 9 in _debug_strnset.c.
Heap state was valid at line 11 of _debug_strnset.c.
```

## **`_debug_strset` — Set characters in a string**

### **Format**

```
#include <string.h>
char *_debug_strset(char *string, size_t c, const char *file, size_t line);
```

### **Description**

This is the debug version of `strset`. Like `strset`, it sets all characters of *string*, except the ending null character (`\0`), to *c* (converted to a char).

`_debug_strset` validates the heap after setting all characters of *string*, and performs this check only when the target is within a heap. `_debug_strset` makes an implicit call to `_heap_check`. If `_debug_strset` detects a corrupted heap when it makes a call to `_heap_check`, `_debug_strset` will report the file name *file* and line number *line* in a message.

### **Return value**

Returns a pointer to the altered string. There is no error return value.

### **Example**

This example contains a programming error. The string, *str*, was created without a null-terminator, and `strset` propagates the letter 'k' until it finds what it thinks is the null-terminator.

```
/* file: _debug_strset.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *str;
    str = (char*)malloc(10);
    strnset(str, 'x', 5);
    strset(str+5, 'k');
    printf("This is the string after strset: %s\n", str);
    return 0;
}
```

The output should be:

```
End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
The first eight bytes of the memory block (in hex) are: 78787878786B6B6B.
This memory block was (re)allocated at line number 9 in _debug_strset.c.
Heap state was valid at line 11 of _debug_strset.c.
Memory error detected at line 12 of _debug_strset.c.
```



---

## 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 L6G 1C7  
Canada

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 document 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 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.

---

## Programming interface information

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

General-use programming interface allow the customer to write application software that obtain 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.

**Note:** 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 the International Business Machines Corporation in the United States, or other countries, or both:

- AIX
- IBM
- IBM (logo)
- PowerPC
- pSeries

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 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 C++ language is also consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:2003 (E)).
- The C and C++ languages are consistent with the OpenMP C and C++ Application Programming Interface Version 2.0.







Program Number: 5724-I11

SC09-7888-00

