IBM XL C for AIX, V10.1

Optimization and Programming Guide

Version 10.1

SC23-8883-00

IBM XL C for AIX, V10.1

Optimization and Programming Guide

Version 10.1

SC23-8883-00

Note

Before using this information and the product it supports, read the information in "Notices" on page 113.

First edition

This edition applies to IBM XL C for AIX, V10.1 (Program number 5724-U80) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Contents

About this information

This guide discusses advanced topics related to the use of the IBM® XL C for AIX®, V10.1 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.

Who should read this information

This document is addressed to programmers building complex applications, who already have experience compiling with XL C, and would like to take further advantage of the compiler's capabilities for program optimization and tuning, support for advanced programming language features, and add-on tools and utilities.

How to use this information

This document uses a ″task-oriented″ approach to presenting the topics, by concentrating on a specific programming or compilation problem in each section. Each topic contains extensive cross-references to the relevant sections of the reference guides in the IBM XL C for AIX, V10.1 documentation set, which provide detailed descriptions of compiler options and pragmas, and specific language extensions.

How this information is organized

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, "Using XL C with Fortran," on page 5 discusses considerations for calling Fortran code from XL C programs.
- Chapter 3, "Aligning data," on page 9 discusses the different compiler options available for controlling the alignment of data in aggregates, such as structures, on all platforms.
- Chapter 4, "Handling floating point operations," on page 19 discusses options available for controlling the way floating-point operations are handled by the compiler.
- Chapter 5, "Using memory heaps," on page 25 discusses compiler library functions for heap memory management, including using custom memory heaps, and validating and debugging heap memory.
- Chapter 6, "Constructing a library," on page 41 discusses how to compile and link static and shared libraries.
- Chapter 7, "Optimizing your applications," on page 45 discusses the various options provided by the compiler for optimizing your programs, and provides recommendations for use of the different options.

- Chapter 9, "Coding your application to improve performance," on page 71 discusses recommended programming practices and coding techniques for enhancing program performance and compatibility with the compiler's optimization capabilities.
- Chapter 10, "Using the high performance libraries," on page 77 discusses two performance libraries that are shipped with XL C: the Mathematical Acceleration Subsystem (MASS), which contains tuned versions of standard math library functions; and the Basic Linear Algebra Subprograms (BLAS), which contains basic functions for matrix multiplication.
- Chapter 11, "Parallelizing your programs," on page 89 provides an overview of the different options offered by the XL C for creating multi-threaded programs, including IBM SMP and OpenMP language constructs.
- Chapter 12, "Memory debug library functions," on page 97 provides a reference listing and examples of all compiler debug memory library functions.

Conventions

Typographical conventions

The following table explains the typographical conventions used in the IBM XL C for AIX, V10.1 information.

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Lowercase commands, executable names, compiler options, and directives.	The compiler provides basic invocation commands, **xlc**, along with several other compiler invocation commands to support various C language levels and compilation environments.
italics	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	Make sure that you update the *size* parameter if you return more than the *size* requested.
<u>underlining</u>	The default setting of a parameter of a compiler option or directive.	nomaf | <u>maf</u>
`monospace`	Programming keywords and library functions, compiler builtins, examples of program code, command strings, or user-defined names.	To compile and optimize myprogram.c, enter: `xlc myprogram.c -O3`.

Syntax diagrams

Throughout this information, diagrams illustrate XL C syntax. This section will help you to interpret and use those 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.

Fragments, which are diagrams of syntactical units other than complete commands, directives, or statements, start with the │─── symbol and end with the ───│ symbol.

- Required items are shown on the horizontal line (the main path):

```
►►──keyword──required_argument────────────────────────────────────►◄
```

- Optional items are shown below the main path:

```
►►──keyword──┬─────────────────────┬──────────────────────────────►◄
             └─optional_argument───┘
```

- If you can choose from two or more items, they are shown vertically, in a stack.

  If you *must* choose one of the items, one item of the stack is shown on the main path.

```
►►──keyword──┬─required_argument1─┬───────────────────────────────►◄
             └─required_argument2─┘
```

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

```
►►──keyword──┬────────────────────┬───────────────────────────────►◄
             ├─optional_argument1─┤
             └─optional_argument2─┘
```

- An arrow returning to the left above the main line (a repeat arrow) indicates that you can make more than one choice from the stacked items or repeat an item. The separator character, if it is other than a blank, is also indicated:

```
             ┌─,───────────────────┐
             ▼                     │
►►──keyword────repeatable_argument─┴──────────────────────────────►◄
```

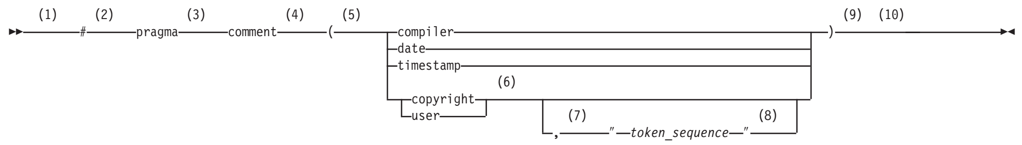
- The item that is the default is shown above the main path.

```
             ┌─default_argument───┐
►►──keyword──┴─alternate_argument─┴───────────────────────────────►◄
```

- Keywords are shown in nonitalic letters and should be entered exactly as shown.
- Variables are shown in italicized lowercase letters. 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.

Sample syntax diagram

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



Notes:

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")
```

Examples in this information

The examples in this information, except where otherwise noted, are coded in a simple style that does not try to conserve storage, check for errors, achieve fast performance, or demonstrate all possible methods to achieve a specific result.

The examples for installation information are labelled as either *Example* or *Basic example*. *Basic examples* are intended to document a procedure as it would be performed during a basic, or default, installation; these need little or no modification.

Related information

The following sections provide related information for XL C:

IBM XL C information

XL C provides product information in the following formats:

- README files

  README files contain late-breaking information, including changes and corrections to the product information. README files are located by default in the XL C directory and in the root directory of the installation CD.
- Installable man pages

Man pages are provided for the compiler invocations and all command-line utilities provided with the product. Instructions for installing and accessing the man pages are provided in the *IBM XL C for AIX , V10.1 Installation Guide*.

- Information center

  The information center of searchable HTML files can be launched on a network and accessed remotely or locally. Instructions for installing and accessing the online information center are provided in the *IBM XL C for AIX , V10.1 Installation Guide*.

  The information center is viewable on the Web at http://publib.boulder.ibm.com/infocenter/comphelp/v101v121/index.jsp.

- PDF documents

  PDF documents are located by default in the /usr/vac/doc/*LANG*/pdf/ directory, where *LANG* is one of en_US, zh_CN, or ja_JP. The PDF files are also available on the Web at http://www.ibm.com/software/awdtools/caix/library.

  The following files comprise the full set of XL C product information:

Table 2. XL C PDF files

Document title	PDF file name	Description
IBM XL C for AIX , V10.1 Installation Guide, GC23-8885-00	install.pdf	Contains information for installing XL C and configuring your environment for basic compilation and program execution.
Getting Started with IBM XL C for AIX , V10.1, GC23-8896-00	getstart.pdf	Contains an introduction to the XL C product, with information on setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
IBM XL C for AIX , V10.1 Compiler Reference, SC23-8882-00	compiler.pdf	Contains information about the various compiler options, pragmas, macros, environment variables, and built-in functions, including those used for parallel processing.
IBM XL C for AIX , V10.1 Language Reference, SC23-8884-00	langref.pdf	Contains information about the C programming languages, as supported by IBM, including language extensions for portability and conformance to nonproprietary standards.
IBM XL C for AIX , V10.1 Optimization and Programming Guide, SC23-8883-00	proguide.pdf	Contains information on advanced programming topics, such as application porting, interlanguage calls with Fortran code, library development, application optimization and parallelization, and the XL C high-performance libraries.

  To read a PDF file, use the Adobe® Reader. If you do not have the Adobe Reader, you can download it (subject to license terms) from the Adobe Web site at http://www.adobe.com.

More information related to XL C including redbooks, white papers, tutorials, and other articles, is available on the Web at:

http://www.ibm.com/software/awdtools/caix/library

Standards and specifications

XL C is designed to support the following standards and specifications. You can refer to these standards for precise definitions of some of the features found in this information.

- *Information Technology – Programming languages – C, ISO/IEC 9899:1990*, also known as *C89*.
- *Information Technology – Programming languages – C, ISO/IEC 9899:1999*, also known as *C99*.
- *Information Technology – Programming languages – Extensions for the programming language C to support new character data types, ISO/IEC DTR 19769*. This draft technical report has been accepted by the C standards committee, and is available at http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1040.pdf.
- *AltiVec Technology Programming Interface Manual*, Motorola Inc. This specification for vector data types, to support vector processing technology, is available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf.
- *Information Technology – Programming Languages – Extension for the programming language C to support decimal floating-point arithmetic, ISO/IEC WDTR 24732*. This draft technical report has been submitted to the C standards committee, and is available at http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1176.pdf.
- *Decimal Types for C++: Draft 4* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1977.html

Other IBM information

- *AIX Commands Reference, Volumes 1 - 6*, *SC23-4888*
- *Technical Reference: Base Operating System and Extensions, Volumes 1 & 2*, *SC23-4913*
- *AIX National Language Support Guide and Reference*, *SC23-4902*
- *AIX General Programming Concepts: Writing and Debugging Programs*, *SC23-4896*
- *AIX Assembler Language Reference*, *SC23-4923*

  All AIX information is available at http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp.
- *Parallel Environment for AIX: Operation and Use*
- *ESSL for AIX V4.2 Guide and Reference*, *SA22-7904*, available at http://publib.boulder.ibm.com/clresctr/windows/public/esslbooks.html

Other information

- *Using the GNU Compiler Collection* available at http://gcc.gnu.org/onlinedocs

Technical support

Additional technical support is available from the XL C Support page at http://www.ibm.com/software/awdtools/caix/support. This page provides a portal with search capabilities to a large selection of Technotes and other support information.

If you cannot find what you need, you can send e-mail to compinfo@ca.ibm.com.

For the latest information about XL C, visit the product information site at http://www.ibm.com/software/awdtools/caix.

How to send your comments

Your feedback is important in helping to provide accurate and high-quality information. If you have any comments about this information or any other XL C information, send your comments by e-mail to compinfo@ca.ibm.com.

Be sure to include the name of the information, the part number of the information, the version of XL C, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

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

You can use XL 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 long and pointer data type sizes and alignment between the two modes. The following table summarizes these differences.

Table 3. 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 .

Related information in the *XL C Compiler Reference*

-q32, -q64

-qwarn64

Compile-time and link-time environment variables

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

Symbolic constant	Mode	Value	Hexadecimal	Decimal
LONG_MIN (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 (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 (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 , 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 >= `UINT_MAX`
- constant < `INT_MIN`
- constant > `INT_MAX`

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

Table 5. 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

Unsuffixed 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 6. Results of bit-shifting long values

Initial value	Symbolic constant	Value after bit shift	
		32-bit mode	64-bit mode
0x7FFFFFFFL	INT_MAX	0xFFFFFFFE	0x00000000FFFFFFFE
0x80000000L	INT_MIN	0x00000000	0x0000000100000000
0xFFFFFFFFL	UINT_MAX	0xFFFFFFFE	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`, when the same code is compiled in 64-bit mode, 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 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 most strictly 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 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.

For detailed information on aligning data structures, including structures that contain bit fields, see Chapter 3, "Aligning data," on page 9.

Calling Fortran code

A significant number of applications use 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 types and the equivalent Fortran types in the different modes.

Table 7. Equivalent C and Fortran data types

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
		integer POINTER (8 bytes)

Related information

Chapter 2, "Using XL C with Fortran," on page 5

Chapter 2. Using XL C with Fortran

With XL C, you can call functions written in Fortran from your C programs. This section discusses some programming considerations for calling Fortran code, in the following areas:

- "Identifiers"
- "Corresponding data types"
- "Character and aggregate data" on page 6
- "Function calls and parameter passing" on page 7
- "Pointers to functions" on page 7
- "Sample program: C calling Fortran" on page 7 provides an example of a C program which calls a Fortran subroutine.

Related information

"Calling Fortran code" on page 4

Identifiers

You should follow these recommendations when writing C code to call functions written in Fortran:

- Avoid using uppercase letters in identifiers. Although XL Fortran folds external identifiers to lowercase by default, the Fortran compiler can be set to distinguish external names by case.
- Avoid using long identifier names. The maximum number of significant characters in XL Fortran identifiers is 250[1].

Corresponding data types

The following table shows the correspondence between the data types available in C and Fortran. Several data types in C have no equivalent representation in Fortran. Do not use them when programming for interlanguage calls.

Table 8. Correspondence of data types among C and Fortran

C data types	Fortran data types
_Bool	LOGICAL(1)
char	CHARACTER
signed char	INTEGER*1
unsigned char	LOGICAL*1
signed short int	INTEGER*2
unsigned short int	LOGICAL*2
signed long int	INTEGER*4
unsigned long int	LOGICAL*4
signed long long int	INTEGER*8
unsigned long long int	LOGICAL*8

1. The Fortran 90 and 95 language standards require identifiers to be no more than 31 characters; the Fortran 2003 standard requires identifiers to be no more than 63 characters.

Table 8. Correspondence of data types among C and Fortran (continued)

C data types	Fortran data types
float	REAL REAL*4
double	REAL*8 DOUBLE PRECISION
long double (default)	REAL*8 DOUBLE PRECISION
long double (with -qlongdouble or -qldbl128)	REAL*16
float _Complex	COMPLEX*8 or COMPLEX(4)
double _Complex	COMPLEX*16 or COMPLEX(8)
long double _Complex (default)	COMPLEX*16 or COMPLEX(8)
long double _Complex(with -qlongdouble or -qldbl128)	COMPLEX*32 or COMPLEX(16)
structure or union	derived type
enumeration	INTEGER*4
char[n]	CHARACTER*n
array pointer to type, or type []	Dimensioned variable (transposed)
pointer to function	Functional parameter
structure (with -qalign=packed)	Sequence derived type

Related information in the *XL C Compiler Reference*

-qldbl128

-qalign

Character and aggregate data

Most numeric data types have counterparts across Cand Fortran. However, character and aggregate data types require special treatment:

- C character strings are delimited by a **'\0'** character. In Fortran, all character variables and expressions have a length that is determined at compile time. Whenever Fortran passes a string argument to another routine, it appends a hidden argument that provides the length of the string argument. This length argument must be explicitly declared in C. The C code should not assume a null terminator; the supplied or declared length should always be used.
- C stores array elements in row-major order (array elements in the same row occupy adjacent memory locations). Fortran stores array elements in ascending storage units in column-major order (array elements in the same column occupy adjacent memory locations). Table 9 shows how a two-dimensional array declared by A[3][2] in C and by A(3,2) in Fortran, is stored:

Table 9. Storage of a two-dimensional array

Storage unit	C element name	Fortran element name
Lowest	A[0][0]	A(1,1)
	A[0][1]	A(2,1)
	A[1][0]	A(3,1)
	A[1][1]	A(1,2)
	A[2][0]	A(2,2)
Highest	A[2][1]	A(3,2)

- In general, for a multidimensional array, if you list the elements of the array in the order they are laid out in memory, a row-major array will be such that the rightmost index varies fastest, while a column-major array will be such that the leftmost index varies fastest.

Function calls and parameter passing

Functions must be prototyped identically in both Cand Fortran.

In C, by default, all function arguments are passed by value, and the called function receives a copy of the value passed to it. In Fortran, by default, arguments are passed by reference, and the called function receives the address of the value passed to it. You can use the Fortran %VAL built-in function or the VALUE attribute to pass by value. Refer to the *XL Fortran Language Reference* for more information.

For call-by-reference (as in Fortran), the address of the parameter is passed in a register. When passing parameters by reference, if you write C functions that call a program written in Fortran, all arguments must be pointers, or scalars with the address operator.

Pointers to functions

A function pointer is a data type whose value is a function address. In Fortran, a dummy argument that appears in an EXTERNAL statement is a function pointer. Function pointers are supported in contexts such as the target of a call statement or an actual argument of such a statement.

Sample program: C calling Fortran

The following example illustrates how program units written in different languages can be combined to create a single program. It also demonstrates parameter passing between C and Fortran subroutines with different data types as arguments.

```
#include <stdio.h>
extern double add(int *, double [], int *, double []);

double ar1[4]={1.0, 2.0, 3.0, 4.0};
double ar2[4]={5.0, 6.0, 7.0, 8.0};

main()
{
int x, y;
double z;

x = 3;
y = 3;


z = add(&x, ar1, &y, ar2); /* Call Fortran add routine */
/* Note: Fortran indexes arrays 1..n */
/* C indexes arrays 0..(n-1) */

printf("The sum of %1.0f and %1.0f is %2.0f \n",
ar1[x-1], ar2[y-1], z);
}
```

The Fortran subroutine is:

```
C Fortran function add.f - for C interlanguage call example

C Compile separately, then link to C program

REAL*8 FUNCTION ADD (A, B, C, D)
REAL*8 B,D
INTEGER*4 A,C
DIMENSION B(4), D(4)
ADD = B(A) + D(C)
RETURN
END
```

Chapter 3. Aligning data

XL 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. In particular, vector types have special alignment requirements which, if not followed, can produce incorrect results. That is, vectors need to be aligned according to a 16 byte boundary. For more information, see the *AltiVec Technology Programming Interface Manual*.

Alignment modes allow you to set alignment defaults for all data types for a compilation unit (or subsection of a compilation unit), by specifying a predefined suboption. Alignment modifiers allow you to set the alignment for specific variables or data types within a compilation unit, by specifying the exact number of bytes that should be used for the alignment.

"Using alignment modes" discusses the default alignment modes for all data types on the different platforms and addressing models; the suboptions and pragmas you can use to change or override the defaults; and rules for the alignment modes for simple variables, aggregates, and bit fields. This section also provides examples of aggregate layouts based on the different alignment modes.

"Using alignment modifiers" on page 15 discusses the different specifiers, pragmas, and attributes you can use in your source code to override the alignment mode currently in effect, for specific variable declarations. It also provides the rules governing the precedence of alignment modes and modifiers during compilation.

Related information in the *XL C Compiler Reference*

-qaltivec

Related external information

AltiVec Technology Programming Interface Manual, available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf

Using alignment modes

Each data type supported by XL C is aligned along byte boundaries according to platform-specific default alignment modes. On AIX, the default alignment mode is **power** or **full**, which are equivalent.

You can change the default alignment mode, by using any of the following mechanisms:

Set the alignment mode for all variables in a single file or multiple files during compilation
To use this approach, you specify the **-qalign** compiler option during compilation, with one of the suboptions listed in Table 10 on page 10.

Set the alignment mode for all variables in a section of source code
To use this approach, you specify the **#pragma align** or **#pragma options align** directives in the source files, with one of the suboptions listed in

Table 10. Each directive changes the alignment mode in effect for all variables that follow the directive until another directive is encountered, or until the end of the compilation unit.

Each of the valid alignment modes is defined in Table 10, which provides the alignment value, in bytes, for scalar variables, for all data types. Where there are differences between 32-bit and 64-bit modes, these are indicated. Also, where there are differences between the first (scalar) member of an aggregate and subsequent members of the aggregate, these are indicated.

Table 10. Alignment settings (values given in bytes)

		Alignment setting				
Data type	**Storage**	**natural**	**power, full**	**mac68k, twobyte**[3]	**bit_packed**[2]	**packed**[2]
_Bool (32-bit mode)	1	1	1	1	1	1
_Bool (64-bit mode)	1	1	1	not supported	1	1
char, signed char, unsigned char	1	1	1	1	1	1
wchar_t (32-bit mode)	2	2	2	2	1	1
wchar_t (64-bit mode)	4	4	4	not supported	1	1
int, unsigned int	4	4	4	2	1	1
short int, unsigned short int	2	2	2	2	1	1
long int, unsigned long int (32-bit mode)	4	4	4	2	1	1
long int, unsigned long int (64-bit mode)	8	8	8	not supported	1	1
_Decimal32	4	4	4	2	1	1
_Decimal64	8	8	8	2	1	1
_Decimal128	16	16	16	2	1	1
long long	8	8	8	2	1	1
float	4	4	4	2	1	1
double	8	8	see note[1]	2	1	1
long double	8	8	see note[1]	2	1	1
long double with **-qldbl128**	16	16	see note[1]	2	1	1
pointer (32-bit mode)	4	4	4	2	1	1
pointer (64-bit mode)	8	8	8	not supported	1	1
vector types	16	16	16	16	1	1

Note:

1. In aggregates, the first member of this data type is aligned according to its natural alignment value; subsequent members of the aggregate are aligned on 4-byte boundaries.
2. The `packed` alignment will not pack bit-field members at the bit level; use the `bit_packed` alignment if you want to pack bit fields at the bit level.
3. For **mac68k** alignment, if the aggregate does not contain a vector member, the alignment is 2 bytes. If an aggregate contains a vector member, then the alignment is the largest alignment of all of its members.

If you are working with aggregates containing `double`, `long long`, or `long double` data types, use the **natural** mode for highest performance, as each member of the aggregate is aligned according to its natural alignment value. If you generate data with an application on one platform and read the data with an application on another platform, it is recommended that you use the **bit_packed** mode, which results in equivalent data alignment on all platforms.

Note: Vectors in a bit-packed structure may not be correctly aligned unless you take extra action to ensure their alignment.

"Alignment of aggregates" discusses the rules for the alignment of entire aggregates and provide examples of aggregate layouts. "Alignment of bit fields" on page 13 discusses additional rules and considerations for the use and alignment of bit fields, and provides an example of bit-packed alignment.

Related information in the *XL C Compiler Reference*

-qalign

-qldbl128

#pragma options

Alignment of aggregates

The data contained in Table 10 on page 10 apply to scalar variables, and variables which are members of aggregates such as structures, unions, and classes. In addition, the following rules apply to aggregate variables, namely structures, unions or classes, as a whole (in the absence of any modifiers):

- For all alignment modes, the size of an aggregate is the smallest multiple of its alignment value that can encompass all of the members of the aggregate.
- Empty aggregates are assigned a size of 0 bytes.
- For all alignment modes except **mac68k**, the alignment of an aggregate is equal to the largest alignment value of any of its members. With the exception of packed alignment modes, members whose natural alignment is smaller than that of their aggregate's alignment are padded with empty bytes.
- For **mac68k** alignment, if the aggregate does not contain a vector member, the alignment is 2 bytes. If an aggregate contains a vector member, then the alignment is the largest alignment of all 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.

The following table shows some examples of the size of an aggregate according to alignment mode.

Table 11. Alignment and aggregate size

	Size of aggregate		
Example	**-qalign=power**	**-qalign=natural**	**-qalign=packed**
struct Struct1 { double a1; char a2; };	16 bytes (The member with the largest alignment requirement is `a1`; therefore, `a2` is padded with 7 bytes.)	16 bytes (The member with the largest alignment requirement is `a1`; therefore, `a2` is padded with 7 bytes.)	9 bytes (Each member is packed to its natural alignment; no padding is added.)

Table 11. Alignment and aggregate size (continued)

Example	Size of aggregate		
	-qalign=power	**-qalign=natural**	**-qalign=packed**

struct Struct2 {
char buf[15];
};
	15 bytes	15 bytes	15 bytes

struct Struct3 {
char c1;
double c2;
};
	12 bytes (The member with the largest alignment requirement is `c2`; however, because it is a `double` and is not the first member, the 4-byte alignment rule applies. `c1` is padded with 3 bytes.)	16 bytes (The member with the largest alignment requirement is `c2`; therefore, `c1` is padded with 7 bytes.)	9 bytes (Each member is packed to its natural alignment; no padding is added.)

For rules on the alignment of aggregates containing bit fields, see "Alignment of bit fields" on page 13.

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=bit_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: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|
```

Alignment of bit fields

You can declare a bit field as a `_Bool`, `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, or `unsigned long long` data type. The alignment of a bit field depends on its base type and the compilation mode (32-bit or 64-bit).

Note: `long long` and `unsigned long long` are not available for C on AIX.

In the C language, you can specify bit fields as `char` or `short` instead of `int`, but XL 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.

However, alignment rules for aggregates containing bit fields are different depending on the alignment mode in effect. 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.
- An aggregate that contains only zero-length bit fields has a length of 0 bytes and an alignment of 4 bytes.

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.
- An aggregate that contains only zero-length bit fields has a length of 0 bytes.

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 zerolength 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-packed alignment

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

Using alignment modifiers

XL C also provides alignment modifiers, which allow you to exercise even finer-grained control over alignment, at the level of declaration or definition of individual variables. Available modifiers are:

#pragma pack(...)

Valid application:
The entire aggregate (as a whole) immediately following the directive. **Note**: on AIX **#pragma pack** does not apply to bit-field union members.

Effect: Sets the maximum alignment of the members of the aggregate to which it applies, to a specific number of bytes. Also allows a bit-field to cross a container boundary. Used to reduce the effective alignment of the selected aggregate.

Valid values:
n: where *n* is 1, 2, 4, 8, or 16. That is, structure members are aligned on *n*-byte boundaries or on their natural alignment boundary, whichever is less. *nopack:* disables packing. *pop*: removes the previous value added with **#pragma pack**. **Note**: empty brackets has the same functionality as *pop*.

__attribute__((aligned(n)))

Valid application:
As a variable attribute, it applies to a single aggregate (as a whole), namely a structure, union, or class; or to an individual member of an aggregate.[1] As a type attribute, it applies to all aggregates declared of that type. If it is applied to a `typedef` declaration, it applies to all instances of that type.[2]

Effect:
Sets the minimum alignment of the specified variable (or variables), to a specific number of bytes. Typically used to increase the effective alignment of the selected variables.

Valid values:
n must be a positive power of 2, or NIL. NIL can be specified as either `__attribute__((aligned()))` or `__attribute__((aligned))`; this is the same as specifying the maximum system alignment (16 bytes on all UNIX® platforms).

__attribute__((packed))

Valid application:
As a variable attribute, it applies to simple variables, or individual members of an aggregate, namely a structure, union or class.[1] As a type attribute, it applies to all members of all aggregates declared of that type.

Effect: Sets the maximum alignment of the selected variable, or variables, to which it applies, to the smallest possible alignment value, namely one byte for a variable and one bit for a bit field.

__align(n)

Effect: Sets the minimum alignment of the variable or aggregate to which it applies to a specific number of bytes; also effectively increases the amount of storage occupied by the variable. Used to increase the effective alignment of the selected variables.

Valid application:
Applies to simple static (or global) variables or to aggregates as a whole, rather than to individual members of aggregates, unless these are also aggregates.

Valid values:
n must be a positive power of 2. XL C also allows you to specify a value greater than the system maximum.

Note:

1. In a comma-separated list of variables in a declaration, if the modifier is placed at the beginning of the declaration, it applies to all the variables in the declaration. Otherwise, it applies only to the variable immediately preceding it.
2. Depending on the placement of the modifier in the declaration of a `struct`, it can apply to the definition of the type, and hence applies to all instances of that type; or it can apply to only a single instance of the type. For details, see *Type Attributes* in the *XL C Language Reference*.

When you use alignment modifiers, the interactions between modifiers and modes, and between multiple modifiers, can become complex. The following sections outline precedence guidelines for alignment modifiers, for the following types of variables:

- simple, or scalar, variables, including members of aggregates (structures, unions or classes) and user-defined types created by `typedef` statements.
- aggregate variables (structures, unions or classes)

Related information in the *XL C Compiler Reference*

#pragma pack

Related information in the *XL C Language Reference*

The aligned type attribute

The packed type attribute

The __align type qualifier

Type attributes

The aligned variable attribute

The packed variable attribute

Guidelines for determining alignment of scalar variables

The following formulas use a "top-down" approach to determining the alignment, given the presence of alignment modifiers, for both non-embedded (stand-alone) scalar variables and embedded scalars (variables declared as members of an aggregate):

Alignment of variable = maximum(*effective type alignment* , *modified alignment value*)

where *effective type alignment* = maximum(maximum(`aligned` type attribute value, `__align` specifier value) , minimum(*type alignment*, `packed` type attribute value))

and *modified alignment value* = maximum(`aligned` variable attribute value, `packed` variable attribute value) and where *type alignment* is the alignment mode currently in effect when the variable is declared, or the alignment value applied to a type in a `typedef` statement.

In addition, for embedded variables, which can be modified by the #pragma pack directive, the following rule applies:

Alignment of variable = maximum(#pragma pack value, maximum(*effective type alignment* , *modified alignment value*))

Note: If a type attribute and a variable attribute of the same kind are both specified in a declaration, the second attribute is ignored.

Guidelines for determining alignment of aggregate variables

The following formulas determine the alignment for aggregate variables, namely structures, unions, and classes:

Alignment of variable = maximum(*effective type alignment* , *modified alignment value*)

where *effective type alignment* = maximum(maximum(`aligned` type attribute value, `__align` specifier value) , minimum(*aggregate type alignment*, `packed` type attribute value))

and *modified alignment value* = maximum (`aligned` variable attribute value , `packed` variable attribute value)

and where *aggregate type alignment* = maximum (alignment of all members)

Note: If a type attribute and a variable attribute of the same kind are both specified in a declaration, the second attribute is ignored.

Chapter 4. Handling floating point operations

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

- "Floating-point formats"
- "Handling multiply-add operations"
- "Compiling for strict IEEE conformance" on page 20
- "Handling floating-point constant folding and rounding" on page 20
- "Handling floating-point exceptions" on page 23

Floating-point formats

XL C supports the following binary floating-point formats:

- 32-bit single precision, with an approximate absolute range and approximate absolute normalized range of 0 and 10^{-38} to 10^{+38} and precision of about 7 decimal digits
- 64-bit double precision, with an approximate absolute range and approximate absolute normalized range of 0 and 10^{-308} to 10^{+308} and precision of about 16 decimal digits
- 128-bit extended precision, with the same range as double-precision values, but with a precision of about 32 decimal digits

Note that the `long double` type may represent either double-precision or extended-precision values, depending on the setting of the **-qldbl128** compiler option. The default is 128 bits. For compatibility with older compilations, you can use **-qnoldbl128** if you need `long double` to be 64 bits.

Beginning in V9.0, on selected hardware and operating system levels, the compiler also supports the following decimal floating-point formats:

- 32-bit single precision, with an approximate range of 10^{-101} to 10^{+90} and precision of 7 decimal digits
- 64-bit double precision, with an approximate range of 10^{-398} to 10^{+369} and precision of 16 decimal digits
- 128-bit extended precision, with an approximate range of 10^{-6176} to 10^{+6111}, and with a precision of 34 decimal digits

Related information in the *XL C Compiler Reference*

-qldbl128

Handling multiply-add operations

By default, the compiler generates a single non-IEEE 754 compatible multiply-add instruction for binary floating-point expressions such as *a+b*c*, partly because one instruction is faster than two. Because no rounding occurs between the multiply and add operations, this may also produce a more precise result. However, the increased precision might lead to different results from those obtained in other environments, and may cause *x*y-x*y* to produce a nonzero result. To avoid these issues, you can suppress the generation of multiply-add instructions by using the **-qfloat=nomaf** option.

Note: Decimal floating-point does not use multiply-add instructions

Related information in the *XL C Compiler Reference*

-qfloat

Compiling for strict IEEE conformance

By default, XL C follows most, but not all of the rules in the IEEE standard. If you compile with the **-qnostrict** option, which is enabled by default at optimization level **-O3** or higher, some IEEE floating-point rules are violated in ways that can improve performance but might affect program correctness. To avoid this issue, and to compile for strict compliance with the IEEE standard, do the following:

- Use the **-qfloat=nomaf** compiler option.
- If the program changes the rounding mode at runtime, use the **-qfloat=rrm** option.
- If the data or program code contains signaling NaN values (NaNS), use the **-qfloat=nans** option. (A signaling NaN is different from a quiet NaN; you must explicitly code it into the program or data or create it by using the **-qinitauto** compiler option.)
- If you compile with **-O3**, **-O4**, or **-O5**, include the option **-qstrict** after it.

Related information

"Advanced optimization" on page 48
Higher optimization levels can have a tremendous impact on performance, but some trade-offs can occur in terms of code size, compilation time, resource requirements, and numeric or algorithmic precision.

Related information in the *XL C Compiler Reference*

-qfloat

-qstrict

-qinitauto

Handling floating-point constant folding and rounding

By default, the compiler replaces most operations involving constant operands with their result at compile time. This process is known as constant folding. Additional folding opportunities may occur with optimization or with the **-qnostrict** option. The result of a floating-point operation folded at compile-time normally produces the same result as that obtained at execution time, except in the following cases:

- The compile-time rounding mode is different from the execution-time rounding mode. By default, both are round-to-nearest; however, if your program changes the execution-time rounding mode, to avoid differing results, do either of the following:
  - Change the compile-time rounding mode to match the execution-time mode, by compiling with the appropriate **-y** option. For more information, and an example, see "Matching compile-time and runtime rounding modes" on page 21.
  - Suppress folding, by compiling with the **-qfloat=nofold** option.
- Expressions like *a+b*c* are partially or fully evaluated at compile-time. The results might be different from those produced at execution time, because *b*c*

might be rounded before being added to *a*, while the runtime multiply-add instruction does not use any intermediate rounding. To avoid differing results, do either of the following:
  - Suppress the use of multiply-add instructions, by compiling with the **-qfloat=nomaf** option.
  - Suppress folding, by compiling with the **-qfloat=nofold** option.
- An operation produces an infinite or NaN result. Compile-time folding prevents execution-time detection of an exception, even if you compile with the **-qflttrap** option. To avoid missing these exceptions, suppress folding with the **-qfloat=nofold** option.

Related information

"Handling floating-point exceptions" on page 23

Related information in the *XL C Compiler Reference*

-qfloat

-qstrict

-qflttrap

Matching compile-time and runtime rounding modes

The default rounding mode used at compile-time and runtime is round-to-nearest, ties even. If your program changes the rounding mode at runtime, the results of a floating-point calculation might be slightly different from those that are obtained at compile-time. The following example illustrates this:[1]

```
#include <float.h>
#include <fenv.h>
#include <stdio.h>

int main ( )
{
 volatile double one = 1.f, three = 3.f;  /* volatiles are not folded */
 double one_third;

 one_third = 1. / 3.;  /* folded */
 printf ("1/3 with compile-time rounding = %.17f\n", one_third);

 fesetround (FE_TOWARDZERO);
 one_third = one / three;  /* not folded */
 fesetround (FE_TONEAREST);[2]
 printf ("1/3 with execution-time rounding to zero = %.17f\n", one_third);

 fesetround (FE_TONEAREST);
 one_third = one / three;  /* not folded */
 fesetround (FE_TONEAREST);[2]
 printf ("1/3 with execution-time rounding to nearest = %.17f\n", one_third);

 fesetround (FE_UPWARD);
 one_third = one / three;  /* not folded */
 fesetround (FE_TONEAREST);[2]
 printf ("1/3 with execution-time rounding to +infinity = %.17f\n", one_third);

 fesetround (FE_DOWNWARD);
 one_third = one / three;  /* not folded */
 fesetround (FE_TONEAREST);[2]
 printf ("1/3 with execution-time rounding to -infinity = %.17f\n", one_third);

 return 0;
 }
```

Note:

1. On AIX, this example must be linked with the system math library, `libm`, to obtain the functions and macros declared in the `fenv.h` header file.
2. See "Rounding modes and standard library functions" for an explanation of the resetting of the round mode before the call to `printf`.

When compiled with the default options, this code produces the following results:

```
1/3 with compile-time rounding = 0.33333333333333331
1/3 with execution-time rounding to zero = 0.33333333333333331
1/3 with execution-time rounding to nearest   = 0.33333333333333331
1/3 with execution-time rounding to +infinity = 0.33333333333333337
1/3 with execution-time rounding to -infinity = 0.33333333333333331
```

Because the fourth computation changes the rounding mode to round-to-infinity, the results are slightly different from the first computation, which is performed at compile-time, using round-to-nearest. If you do not use the **-qfloat=nofold** option to suppress all compile-time folding of floating-point computations, it is recommended that you use the **-y** compiler option with the appropriate suboption to match compile-time and runtime rounding modes. In the previous example, compiling with **-yp** (round-to-infinity) produces the following result for the first computation:

```
1/3 with compile-time rounding = 0.33333333333333337
```

In general, if the rounding mode is changed to +infinity or -infinity, or to any decimal floating-point only rounding mode, it is recommended that you also use the **-qfloat=rrm** option.

Related information in the *XL C Compiler Reference*

-qfloat

-y

Rounding modes and standard library functions

On AIX, C input/output and conversion functions apply the rounding mode in effect to the values that are input or output by the function. These functions include `printf`, `scanf`, `atof`, and `ftoa`,.

For example, if the current rounding mode is round-to-infinity, the `printf` function will apply that rounding mode to the floating-point digit string value it prints, in addition to the rounding that was already performed on a calculation. The following example illustrates this:

```
#include <float.h>
#include <fenv.h>
#include <stdio.h>

int main( )
{
 volatile double one = 1.f, three = 3.f;  /* volatiles are not folded*/
 double one_third;

 fesetround (FE_UPWARD);
 one_third = one / three;  /* not folded */
 printf ("1/3 with execution-time rounding to +infinity = %.17f\n", one_third);

 fesetround (FE_UPWARD);
 one_third = one / three;  /* not folded */
 fesetround (FE_TONEAREST);
 printf ("1/3 with execution-time rounding to +infinity = %.17f\n", one_third);

 return 0;
}
```

When compiled with the default options, this code produces the following results:

```
1/3 with execution-time rounding to +infinity = 0.33333333333333338
1/3 with execution-time rounding to -infinity = 0.33333333333333337
```

In the first calculation, the value returned is rounded upward to 0.33333333333333337, but the `printf` function rounds this value upward again, to print out 0.33333333333333338. The solution to this problem, which is used in the second calculation, is to reset the rounding mode to round-to-nearest just before the call to the library function is made.

Handling floating-point exceptions

By default, invalid operations such as division by zero, division by infinity, overflow, and underflow are ignored at runtime. 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 runtime will not occur. To ensure that the **-qflttrap** option traps all runtime floating-point exceptions, consider using the **-qfloat=nofold** option to suppress all compile-time folding.

Related information in the *XL C Compiler Reference*

-qfloat

-qflttrap

Compiling a decimal floating-point program

If you are using decimal floating-point formats in your programs, use the **-qdfp** option when you compile them. For example, to compile the following Hello World program dfp_hello.c, the compiler invocation would be:

```
xlc -qdfp dfp_hello.c
```

```
#include <stdio.h>
#include <float.h>
int main() {
   printf("Hello DFP World\n");
   printf("DEC32_MAX = %Hf\n",DEC32_MAX);
   float f = 12.34df;
   printf("12.34df as a float = %f\n",f);
 }
```

Related information in the *XL C Compiler Reference*

-qdfp

Chapter 5. Using memory heaps

In addition to the memory management functions defined by ANSI, XL 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 runtime 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 36 discusses the functions available for checking and debugging the default and user-created heaps.

Managing memory with multiple heaps

You can use XL C to create and manipulate your own memory heaps, either in place of or in addition to the default XL C runtime 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 runtime 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 runtime 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 only 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 runtime 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.

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 12. Functions for managing memory heaps

Default heap function	**Corresponding user-created heap function**	**Description**
n/a	_ucreate	Creates a heap. Described in “Creating a heap” on page 27.
n/a	_uopen	Opens a heap for use by a process. Described in “Using a heap” on page 29.
n/a	_ustats	Provides information about a heap. Described in “Getting information about a heap” on page 30.
n/a	_uaddmem	Adds memory blocks to a heap. Described in “Expanding a heap” on page 28.
n/a	_uclose	Closes a heap from further use by a process. Described in “Closing and destroying a heap” on page 30.
n/a	_udestroy	Destroys a heap. Described in “Closing and destroying a heap” on page 30.
calloc	_ucalloc	Allocates and initializes memory from a heap you have created. Described in “Using a heap” on page 29.
malloc	_umalloc	Allocates memory from a heap you have created. Described in “Using a heap” on page 29.
_heapmin	_uheapmin	Returns unused memory to the system. Described in “Closing and destroying a heap” on page 30.

Table 12. Functions for managing memory heaps (continued)

Default heap function	**Corresponding user-created heap function**	**Description**
n/a	_udefault	Changes the default runtime heap to a user-created heap. Described in "Changing the default heap used in a program" on page 31.

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.

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).

Use _HEAP_REGULAR for regular memory. Most programs use regular memory. This is the type provided by the default run-time heap. Use _HEAP_SHARED for shared memory. Heaps of shared memory can be shared between processes or applications.

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

Creating a dynamically-sized heap

With the XL C default heap, when not enough storage is available to fulfill a malloc request, the runtime environment gets additional storage from the system. Similarly, when you minimize the heap with _heapmin or when your program ends, the runtime 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.

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 applications is the runtime heap provided by XL 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 runtime 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 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 default runtime 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 runtime 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 progc.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);
   _ufree(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                 */
       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 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 runtime 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 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 37.)
- Debug versions of all memory management functions. (Described in “Functions for debugging memory heaps” on page 37.)

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 information

Chapter 12, "Memory debug library functions," on page 97

Related information in the *XL C Compiler Reference*

-qheapdebug

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 13. Functions for checking memory heaps

Default heap function	**User-created heap function**	**Description**
_heapchk	_uheapchk	Checks the entire heap for minimal consistency.
_heapset	_uheapset	Checks the free memory in the heap for minimal consistency, and sets the free memory in the heap to a value you specify.
_heap_walk	_uheap_walk	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 31.

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*.

Table 14. Functions for debugging memory heaps

Default heap function	Corresponding user-created heap function
_debug_calloc	_debug_ucalloc
_debug_malloc	_debug_umalloc
_debug_heapmin	_debug_uheapmin
_debug_realloc	n/a
_debug_free	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 31.

Note:

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 15. Additional functions for debugging memory heaps

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

Table 15. Additional functions for debugging memory heaps (continued)

Default heap function	Corresponding user-created heap function	Description
_heap_check	_uheap_check	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.

Related information

Chapter 12, "Memory debug library functions," on page 97

Related information in the *XL C Compiler Reference*

-brtl

-qheapdebug

-qro / #pragma strings

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 use the `HD_SKIP` environment variable to control how often the functions check the heap. 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 6. Constructing a library

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

Related information

"Compiling and linking a library"

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. For example:
   ```
   xlc -c bar.c example.c
   ```
2. Use the ar command to add the generated object files to an archive library file. For example:
   ```
   ar -rv libfoo.a bar.o example.o
   ```

Compiling a shared library

To compile 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 42.
   - 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. 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.)
4. 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
   ```
5. Link the shared library to the main application, as described in "Linking a library to an application" on page 43.

To create a shared library that uses runtime linking:

1. Follow steps 1 and 2 in the procedure described above.

2. 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. For example:

   ```
   xlc -G -o libfoo.so foo1.o foo2.o -bE:exportlist
   ```
3. Link the shared library to the main application, as described in "Linking a library to an application" on page 43.

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──┬─-f─file_list─┬──┬────┬──┬───────────────┬──────►◄
                     └─-r─┘            └─obj_files────┘  └─-w─┘  │      ┌─32─┐   │
                                                                 └─-X───┴─64─┴───┘
```

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.

-w
: Excludes weak symbols from the export list.

-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*.

The **CreateExportList** command creates an empty list if any of the following are true:
- No object files are specified by either **-f** *file_list* or *obj_files*.
- The file specified by the **-f** *file_list* parameter is empty.

Related external information
- **ar** and **ld** in the *AIX Commands Reference, Volumes 1 - 6*

  Related information in the *XL C Compiler Reference*

  -qexpfile

  -qmkshrobj

  -O, -qoptimize

-G

-brtl

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 runtime 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.

Related information in the *XL C Compiler Reference*

-l

-L

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 -o mylib.so myfile.o -Ldirectory -lfoo
```

Related information in the *XL C Compiler Reference*

-qmkshrobj

-L

Chapter 7. Optimizing your applications

The XL compilers enable development of high performance 32-bit and 64-bit applications by offering a comprehensive set of performance enhancing techniques that exploit the multilayered PowerPC® architecture. These performance advantages depend on good programming techniques, thorough testing and debugging, followed by optimization, and tuning.

Distinguishing between optimization and tuning

You can use optimization and tuning separately or in combination to increase the performance of your application. Understanding the difference between them is the first step in understanding how the different levels, settings and techniques can increase performance.

Optimization

Optimization is a compiler driven process that searches for opportunities to restructure your source code and give your application better overall performance at runtime, without significantly impacting development time. The XL compiler optimization suite, which you control using compiler options and directives, performs best on well-written source code that has already been through a thorough debugging and testing process. These optimization transformations can:

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

Consider that although not all optimizations benefit all applications, even basic optimization techniques can result in a performance benefit. Consult the Steps in the optimization process for an overview of the common sequence of steps you can use to increase the performance of your application.

Tuning

Where optimization applies increasingly aggressive transformations designed to improve the performance of any application in any supported environment, tuning offers you opportunities to adjust characteristics of your application to improve performance, or to target specific execution environments. Even at low optimization levels, tuning for your application and target architecture can have a positive impact on performance. With proper tuning the compiler can:

- Select more efficient machine instructions.
- Generate instruction sequences that are more relevant to your application.

Steps in the optimization process

As you begin the optimization process, consider that not all optimization techniques suit all applications. Trade-offs sometimes occur between an increase in compile time, a reduction in debugging capability, and the improvements that optimization can provide.

Learning about, and experimenting with different optimization techniques can help you strike the right balance for your XL compiler applications while achieving the best possible performance. Also, though it is unnecessary to hand-optimize your code, compiler-friendly programming can be extremely beneficial to the optimization process. Unusual constructs can obscure the characteristics of your application and make performance optimization difficult. Use the steps in this section as a guide for optimizing your application.

1. The Basic optimization step begins your optimization processes at levels 0 and 2.
2. The Advanced optimization step exposes your application to more intense optimizations at levels 3 through 5.
3. The Debugging high-performance code step can help you identify issues and problems that can occur with optimized code.

Basic optimization

The XL compiler supports several levels of optimization, with each option level building on the levels below through increasingly aggressive transformations, and consequently using more machine resources.

Ensure that your application compiles and executes properly at low optimization levels before trying more aggressive optimizations. This topic discusses two optimizations levels, listed with complementary options in the *Basic optimizations* table. The table also includes a column for compiler options that can have a performance benefit at that optimization level for some applications.

Table 16. Basic optimizations

Optimization level	Additional options implied by default	Complementary options	Other options with possible benefits
-O0	None	**-qarch**	**-g**
-O2	**-qmaxmem=**8192	**-qarch** **-qtune**	**-qmaxmem=**-1 **-qhot=level=0**

Optimizing at level 0

Benefits at level 0

- Minimal performance improvement, with minimal impact on machine resources.
- Exposes some source code problems, helping in the debugging process.

Begin your optimization process at **-O0** which the compiler already specifies by default. For SMP programs, the closest equivalent to **-O0** is **-qsmp=noopt**. This level performs basic analytical optimization by removing obviously redundant code, and can result in better compile time, while ensuring your code is algorithmically correct so you can move forward to more complex optimizations. **-O0** also includes constant folding. The option **-qfloat=nofold** can be used to suppress folding floating-point operations. Optimizing at this level accurately preserves all debug information and can expose problems in existing code, such as uninitialized variables.

Additionally, specifying **-qarch** at this level targets your application for a particular machine and can significantly improve performance by ensuring your application takes advantage of all applicable architectural benefits.

Optimizing at level 2

Benefits at level 2

- Eliminates redundant code
- Basic loop optimization
- Can structure code to take advantage of **-qarch** and **-qtune** settings

After successfully compiling, executing, and debugging your application using **-O0**, recompiling at **-O2** opens your application to a set of comprehensive low-level transformations that apply to subprogram or compilation unit scopes and can include some inlining. Optimizations at **-O2** are a relative balance between increasing performance while limiting the impact on compilation time and system resources. You can increase the memory available to some of the optimizations in the **-O2** portfolio by providing a larger value for the **-qmaxmem** option. Specifying **-qmaxmem=-1** allows the optimizer to use memory as needed without checking for limits but does not change the transformations the optimizer applies to your application at **-O2**.

Starting to tune at O2

Choosing the right hardware architecture target or family of targets becomes even more important at **-O2** and higher. Targeting the proper hardware allows the optimizer to make the best use of the hardware facilities available. If you choose a family of hardware targets, the **-qtune** option can direct the compiler to emit code consistent with the architecture choice, but will execute optimally on the chosen tuning hardware target. This allows you to compile for a general set of targets but have the code run best on a particular target.

The **-O2** option can perform a number of additional optimizations, including:

- Common subexpression elimination: Eliminates redundant instructions.
- Constant propagation: Evaluates constant expressions at compile-time.
- Dead code elimination: Eliminates instructions that a particular control flow does not reach, or that generate an unused result.
- Dead store elimination: Eliminates unnecessary variable assignments.
- Graph coloring register allocation: Globally assigns user variables to registers.
- Value numbering: Simplifies algebraic expressions, by eliminating redundant computations.
- Instruction scheduling for the target machine.
- Loop unrolling and software pipelining.
- Moves invariant code out of loops.
- Simplifies control flow.
- Strength reduction and effective use of addressing modes.

Even with **-O2** optimizations, some useful information about your source code is made available to the debugger if you specify **-g**. Conversely, higher optimization levels can transform code to an extent to which debug information is no longer accurate. Use that information with discretion.

Advanced optimization

Higher optimization levels can have a tremendous impact on performance, but some trade-offs can occur in terms of code size, compilation time, resource requirements, and numeric or algorithmic precision.

After applying basic optimizations and successfully compiling and executing your application, you can apply more powerful optimization tools. The XL compiler optimization portfolio includes many options for directing advanced optimization, and the transformations your application undergoes are largely under your control. The discussion of each optimization level in Table 17 includes information on not only the performance benefits, and the possible trade-offs as well, but information on how you can help guide the optimizer to find the best solutions for your application.

Table 17. Advanced optimizations

Optimization Level	Additional options implied	Complementary options	Options with possible benefits
-O3	**-qnostrict** **-qmaxmem=-1** **-qhot=level=0**	**-qarch** **-qtune**	**-qpdf**
-O4	**-qnostrict** **-qmaxmem=-1** **-qhot** **-qipa** **-qarch=auto** **-qtune=auto** **-qcache=auto**	**-qarch** **-qtune** **-qcache**	**-qpdf** **-qsmp=auto**
-O5	All of **-O4** **-qipa=level=2**	**-qarch** **-qtune** **-qcache**	**-qpdf** **-qsmp=auto**

Optimizing at level 3

Benefits at level 3

- Better loop scheduling
- High-order loop analysis and transformations (**-qhot=level=0**)
- Inlining of small procedures within a compilation unit by default
- Eliminating implicit compile-time memory usage limits
- Widening, which merges adjacent load/stores and other operations
- Pointer aliasing improvements to enhance other optimizations

Specifying **-O3** initiates more intense low-level transformations that remove many of the limitations present at **-O2**. For instance, the optimizer no longer checks for memory limits, by defaulting to **-qmaxmem=-1**. Additionally, optimizations encompass larger program regions and attempt more in-depth analysis. While not all applications contain opportunities for the optimizer to provide a measurable increase in performance, most applications can benefit from this type of analysis.

Potential trade-offs at level 3

With the in-depth analysis of **-O3** comes a trade-off in terms of compilation time and memory resources. Also, since **-O3** implies **-qnostrict**, the optimizer can alter certain floating-point semantics in your application to gain execution speed. This typically involves precision trade-offs as follows:

- Reordering of floating-point computations.
- Reordering or elimination of possible exceptions, such as division by zero or overflow.

You can still gain most of the **-O3** benefits while preserving precise floating-point semantics by specifying **-qstrict**. Compiling with **-qstrict** is necessary if you require the same absolute precision in floating-point computational accuracy as you get with **-O0**, **-O2**, or **-qnoopt** results. The option **-qstrict=ieeefp** also ensures adherence to all IEEE semantics for floating-point operations. If your application is sensitive to floating-point exceptions or the order of evaluation for floating-point arithmetic, compiling with **-qstrict**, **-qstrict=exceptions**, or **-qstrict=order**, helps to ensure accurate results. You should also consider the impact of the **-qstrict=precision** suboption group on floating-point computational accuracy. The precision suboption group includes the individual suboptions: **subnormals**, **operationprecision**, **association**, **reductionorder**, and **library** (described in the **-qstrict** option in the *XL C Compiler Reference*).

Without **-qstrict**, the difference in computation for any one source-level operation is very small in comparison to basic optimization. Although a small difference can be compounded if the operation is in a loop structure where the difference becomes additive, most applications are not sensitive to the changes that can occur in floating-point semantics.

See the -O option in the *XL C Compiler Reference* for information on the **-O** level syntax.

An intermediate step: adding -qhot suboptions at level 3

At **-O3**, the optimization includes minimal **-qhot** loop transformations at **level=0** to increase performance. You can further increase your performance benefit by increasing the level and therefore the aggressiveness of **-qhot**. Try specifying **-qhot** without any suboptions, or **-qhot=level=1**.

Optimizing at level 4

Benefits at level 4

- Propagation of global and argument values between compilation units
- Inlining code from one compilation unit to another
- Reorganization or elimination of global data structures
- An increase in the precision of aliasing analysis

Optimizing at **-O4** builds on **-O3** by triggering **-qipa=level=1** which performs interprocedural analysis (IPA), optimizing your entire application as a unit. This option is particularly pertinent to applications that contain a large number of frequently used routines.

To make full use of IPA optimizations, you must specify **-O4** on the compilation and link steps of your application build as interprocedural analysis occurs in stages at both compile and link time.

Potential trade-offs at level 4

In addition to the trade-offs already mentioned for **-O3**, specifying **-qipa** can significantly increase compilation time, especially at the link step.

The IPA process

1. At compilation time optimizations occur on a file-by-file basis, as well as preparation for the link stage. IPA writes analysis information directly into the object files the compiler produces.
2. At the link stage, IPA reads the information from the object files and analyzes the entire application.
3. This analysis guides the optimizer on how to rewrite and restructure your application and apply appropriate **-O3** level optimizations.

Beyond **-qipa**, **-O4** enables other optimization options:

- **-qhot**

  Enables more aggressive HOT transformations to optimize loop constructs and array language.
- **-qarch=**auto and **-qtune=**auto

  Optimizes your application to execute on a hardware architecture identical to your build machine. If the architecture of your build machine is incompatible with your application's execution environment, you must specify a different **-qarch** suboption after the **-O4** option. This overrides **-qarch=auto**.
- **-qcache=auto**

  Optimizes your cache configuration for execution on specific hardware architecture. The auto suboption assumes that the cache configuration of your build machine is identical to the configuration of your execution architecture. Specifying a cache configuration can increase program performance, particularly loop operations by blocking them to process only the amount of data that can fit into the data cache.

  If you will be executing your application on a different machine, specify correct cache values.

Optimizing at level 5

Benefits at level 5

- Most aggressive optimizations available

As the highest optimization level, **-O5** includes all **-O4** optimizations and deepens whole program analysis by increasing the **-qipa** level to 2. Compiling with **-O5** also increases how aggressively the optimizer pursues aliasing improvements. Additionally, if your application contains a mix of C/C++ and Fortran code that you compile using XL compilers, you can increase performance by compiling and linking your code with the **-O5** option.

Potential trade-offs at level 5

Compiling at **-O5** requires more compilation time and machine resources than any other optimization level, particularly if you include **-O5** on the IPA link step. Compile at **-O5** as the final phase in your optimization process after successfully compiling and executing your application at **-O4**.

Tuning for your 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 18. Target machine options

Option	Behavior
-q32	Generates code for a 32-bit (4 byte integer / 4 byte long / 4 byte pointer) addressing model (32-bit execution mode). This is the default setting.
-q64	Generates code for a 64-bit (4 byte integer / 8 byte long / 8 byte pointer) addressing model (64-bit execution mode).
-qarch	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 architecture. Using -O4 or -O5 sets the default to -qarch=auto. See "Getting the most out of target machine options" on page 52 below for more information on this option.
-qipa=clonearch	Allows you to specify multiple specific processor architectures for which instruction sets will be generated. At runtime, the application will detect the specific architecture of the operating environment and select the instruction set specialized for that architecture. The advantage of this option is that it allows you to optimize for several architectures without recompiling your code for each target architecture. See "Using interprocedural analysis" on page 56 for more information on this option.
-qtune	Biases optimization toward execution on a given microprocessor, without implying anything about the instruction set architecture to use as a target. See "Getting the most out of target machine options" on page 52 below for more information on this option.
-qcache	Defines a specific cache or memory geometry. The defaults are determined through the setting of -qtune. See "Getting the most out of target machine options" on page 52 below for more information on this option.

For a complete listing of valid hardware-related suboptions and combinations of suboptions, see *Acceptable -qarch/-qtune combinations* in the **-qtune** section of the *XL C Compiler Reference* and see *Specifying Compiler Options for Architecture-Specific, 32- or 64-bit Compilation* in the *XL C Compiler Reference*.

Related information in the *XL C Compiler Reference*

- -q32, -q64
- -qarch
- -qipa
- -qtune
- -qcache

Specifying compiler options for architecture-specific, 32-bit or 64-bit compilation

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, or use the **-qipa=clonearch** option, which will generate instructions for multiple architectures. Note that if you use **-qipa=clonearch**, the **-qarch** value must be in the family of architectures specified by the **clonearch** suboption.

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 square root 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™, POWER6™, 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 -qarch/-qtune combinations* in the **-qtune** section of the *XL C Compiler Reference*.

If you need to create a single binary that will run on a range of PowerPC hardware, consider using the **-qtune=balanced** option. With this option in effect, optimization decisions made by the compiler are not targeted to a specific version of hardware. Instead, tuning decisions try to include features that are generally helpful across a broad range of hardware and avoid those optimizations that may be harmful on some hardware. Note that you should verify the performance of code compiled with the **-qtune=balanced** option before distributing it.

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 the *XL C Compiler Reference*.

Related information in the *XL C Compiler Reference*

- -qignerrno
- -qhot
  - -qsmp
  - -qcache
  - -qlistopt
  - -qarch
  - -qtune

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.
- Generating vector instructions.

To enable high-order loop analysis and transformations, you use the **-qhot** option, which implies an optimization level of **-O2**. The following table lists the suboptions available for **-qhot**.

Table 19. -qhot suboptions

Suboption	Behavior
level=1	This is the default suboption if you specify -qhot with no suboptions. This level is also automatically enabled if you compile with -O4 or -O5. This is equivalent to specifying -qhot=vector and -qhot=simd.
level=0	Instructs the compiler to perform a subset of high-order transformations that enhance performance by improving data locality. This suboption implies -qhot=novector, -qhot=noarraypad and -qhot=nosimd. This level is automatically enabled if you compile with -O3.
vector	When specified with **-qnostrict** and **-qignerrno**, or -O3 or a higher optimization level, instructs the compiler to transform some loops to use the optimized versions of various math functions contained in the MASS libraries, rather than use the system versions. The optimized versions make different trade-offs with respect to accuracy and exception-handling versus performance. This suboption is enabled by default if you specify -qhot with no suboptions. Also, specifying -qhot=vector with -O3 implies -qhot=level=1.
arraypad	Instructs the compiler to pad any arrays where it infers there might be a benefit and to pad by whatever amount it chooses.

Table 19. -qhot suboptions (continued)

Suboption	**Behavior**
simd	Instructs the compiler to attempt automatic SIMD vectorization; that is, converting certain operations in a loop that apply to successive elements of an array into a call to a VMX instruction. This call calculates several results at one time, which is faster than calculating each result sequentially. This suboption is enabled on AIX if you set -qarch to ppc970 or pwr6 and use **-qenablevmx**.

Related information in the *XL C Compiler Reference*

-qhot

-qstrict

-qignerrno

-qarch

-qenablevmx

Getting the most out of -qhot

Here are some suggestions for using -qhot:

- Try using -qhot along with -O3 for all of your code. It is designed to have a neutral effect when no opportunities for transformation exist.
- If the runtime performance of your code can significantly benefit from automatic inlining and memory locality optimizations, try using -O4 with -qhot=level=0 or -qhot=novector.
- If you encounter unacceptably long compile times (this can happen with complex loop nests), try -qhot=level=0.
- If your code size is unacceptably large, try using **-qcompact** along with -qhot.
- If necessary, deactivate -qhot selectively, allowing it to improve some of your code.
- Use -qreport along with -qhot=simd to generate a loop transformation listing. The listing file identifies how loops were transformed in a section marked `LOOP TRANSFORMATION SECTION`. Use the listing information as feedback about how the loops in your program are being transformed. Based on this information, you may want to adjust your code so that the compiler can transform loops more effectively. For example, you can use this section of the listing to identify non-stride-one references that may prevent loop vectorization.

Related information in the *XL C Compiler Reference*

-qcompact

-qhot

-qenablevmx

-qstrict

Using shared-memory parallelism (SMP)

Many 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 **-qsmp** in the *XL C Compiler Reference*. An overview of automatic parallelization, as well as of IBM SMP and OpenMP directives is provided in Chapter 11, "Parallelizing your programs," on page 89.

Table 20. Commonly used -qsmp suboptions

suboption	Behavior
auto	Instructs the compiler to automatically generate parallel code where possible without user assistance. Any SMP programming constructs in the source code, including IBM SMP and OpenMP directives, are also recognized. This is the default setting if you do not specify any **-qsmp** suboptions, and it also implies the **opt** suboption.
omp	Instructs the compiler to enforce strict conformance to the OpenMP API for specifying explicit parallelism. Only language constructs that conform to the OpenMP standard are recognized. Note that **-qsmp=omp** is currently incompatible with **-qsmp=auto**.
opt	Instructs the compiler to optimize as well as parallelize. The optimization is equivalent to **-O2 -qhot** in the absence of other optimization options.
noopt	All optimization is turned off. During development, it can be useful to turn off optimization to facilitate debugging.
fine_tuning	Other values for the suboption provide control over thread scheduling, nested parallelism, locking, etc.

Related information in the *XL C Compiler Reference*

-O, -qoptimize

-qsmp

-qhot

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 runtime environment uses all available processors. Do not set the *XLSMPOPTS=PARTHDS* 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 variable) 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 information in the *XL C Compiler Reference*

-qsmp

-qhot

Invoking the compiler

XLSMPOPTS

Environment variables for parallel processing

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 in whole program mode (with the exception of the **clonearch** and **cloneproc** suboptions, which must be specified on the link step). 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 the **-qipa** section of the *XL C Compiler Reference*.

The steps to use IPA are:

1. Do preliminary performance analysis and tuning before compiling with the **-qipa** option, because the IPA analysis uses a two-pass mechanism that increases compile and link time. You can reduce some compile and link overhead by using the **-qipa=noobject** option.
2. Specify the **-qipa** option on both the compile and the link steps of the entire application, or as much of it as possible. Use suboptions to indicate assumptions to be made about parts of the program *not* compiled with **-qipa**.

Table 21. Commonly used **-qipa** *suboptions*

Suboption	Behavior
level=0	Program partitioning and simple interprocedural optimization, which consists of: • Automatic recognition of standard libraries. • Localization of statically bound variables and procedures. • Partitioning and layout of procedures according to their calling relationships. (Procedures that call each other frequently are located closer together in memory.) • Expansion of scope for some optimizations, notably register allocation.

Table 21. Commonly used ***-qipa*** *suboptions (continued)*

Suboption	**Behavior**
level=1	Inlining and global data mapping. Specifically: • Procedure inlining. • Partitioning and layout of static data according to reference affinity. (Data that is frequently referenced together will be located closer together in memory.) This is the default level if you do not specify any suboptions with the -qipa option.
level=2	Global alias analysis, specialization, interprocedural data flow: • 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. • Intensive intraprocedural optimizations. This can take the form of value numbering, code propagation and simplification, moving code into conditions or out of loops, and elimination of redundancy. • Interprocedural constant propagation, dead code elimination, pointer analysis, code motion across functions, and interprocedural strength reduction. • Procedure specialization (cloning). • Whole program data reorganization.
inline=*suboptions*	Allows precise control over function inlining.
clonearch=*arch_list*	Allows you to specify multiple architectures for which optimized instructions can be generated. Supported architecture values are PWR4, PWR5, PWR6, and PPC970. For every function in your program, the compiler generates a generic version of the instruction set, according to the -qarch value in effect, and, if appropriate, clones specialized versions of the instruction set for the architectures you specify in this suboption. The compiler inserts code into your application to check for the processor architecture at run time, and selects the version of the generated instructions that is optimized for the runtime environment.
cloneproc=*func_list*	Allows you to specify the exact functions which should be cloned for the specified architectures in the **clonearch** suboption.
fine_tuning	Other values for **-qipa** provide the ability to specify the behavior of library code, tune program partitioning, read commands from a file, etc.

Related information in the *XL C Compiler Reference*

-qipa

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. Although you can also use **-qipa** with libraries, shared objects, and executable files, 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). You can use the TMPDIR environment variable to specify a 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 **-qipa=list=long** 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 **-qipa=[no]inline=***function_name*.

Note: While IPA's interprocedural optimizations can significantly improve performance of a program, they can also cause incorrect but previously functioning programs to fail. Here are examples of programming practices that can work by accident without aggressive optimization but are exposed with IPA:
- Relying on the allocation order or location of automatic variables, such as taking the address of an automatic variable and then later comparing it with the address of another local variable to determine the growth direction of a stack. The C language does not guarantee where an automatic variable is allocated, or its position relative to other automatic variables. Do not compile such a function with IPA.
- Accessing a pointer that is either invalid or beyond an array's bounds. Because IPA can reorganize global data structures, a wayward pointer which might have previously modified unused memory might now conflict with user-allocated storage.

Related information in the *XL C Compiler Reference*

-Q, -qinline

-qlist

-qipa

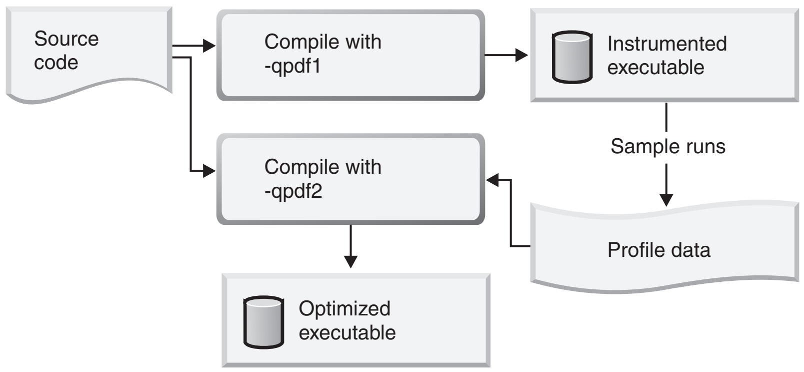
Using profile-directed feedback

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.

The PDF process is intended to be used after other debugging and tuning is finished, as one of the last steps before putting the application into production. Other optimizations such as **-qipa** and optimization levels **-O4** and **-O5** can also benefit when used in conjunction with PDF.

The following diagram illustrates the PDF process.

Figure 1. Profile-directed feedback



You first compile the program with the **-qpdf1** option (with a minimum optimization level of **-O2**), 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. Alternatively, if you want to save considerable time by avoiding a full recompilation in the **-qpdf2** step, you can simply relink the object files produced by the **-qpdf1** step.

To use PDF, follow these steps:

1. Compile some or all of the source files in a program with the **-qpdf1** option. You need to specify at least the **-O2** optimizing option and you also need to link with at least **-O2** in effect. Note the compiler options that you use to compile the files; you will need to use the same options later.
2. Run the program all the way through using data that is representative of the data that will be used during a normal run of your finished program. The program records profiling information when it finishes. You can run the program multiple times with different data sets, and the profiling information is accumulated to provide a count of how often branches are taken and blocks of code are executed, based on the input data used. When the application exits, by default, it writes profiling information to the PDF file in the current working directory or the directory specified by the PDFDIR environment variable. The default name for the instrumentation file is ._pdf . To override the defaults, use the **-qipa=pdfname** option in the **-qpdf1** step.
3. Recompile your program using the same compiler options as before, but change **-qpdf1** to **-qpdf2**. In this second compilation, the accumulated profiling information is used to fine-tune the optimizations. The resulting program contains no profiling overhead and runs at full speed.

   Note: The options **-L**, **-l**, and some others are linker options, and you can change them at this point.

As an intermediate step, you can use **-qpdf2** to link the object files created by the **-qpdf1** pass without recompiling the source on the **-qpdf2** pass. This can save considerable time and help fine tune large applications for optimization. You can create and test different flavors of PDF optimized binaries by using different options on the **-qpdf2** pass.

Notes:

- 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.

- When compiling your program with **-qpdf1** or **-qpdf2**, by default, the **-qipa** option is also invoked with **level=0**
- To avoid wasting compilation and execution time, make sure that the PDFDIR environment variable is set to an absolute path. Otherwise, you might run the application from the wrong directory, and it will not be able to locate the profile data files. When that happens, the program may not be optimized correctly or may be stopped by a segmentation fault. A segmentation fault might also happen if you change the value of the PDFDIR variable and execute the application before finishing the PDF process.
- You must use the same set of compiler options at all compilation steps for a particular program. Otherwise, PDF cannot optimize your program correctly and may even slow it down. All compiler settings must be the same, including any supplied by configuration files.
- Avoid mixing PDF files created by the current version level of XL C with PDF files created by other version levels of the compiler.
- If you compile a program with **-qpdf1**, remember that it will generate profiling information when it runs, which involves some performance overhead. This overhead goes away when you recompile with **-qpdf2** or with no PDF at all.

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** and a minimum of **-O2**.
2. Run the application using a typical data set or several typical data sets. By default, this produces a PDF file in the current directory. The default name of the PDF file is ._pdf.
3. Change the PDF file location specified by the PDFDIR environment variable or the **-qipa=pdfname** option to produce a PDF file in a different location.
4. Recompile or relink the application with **-qpdf1** and a minimum of **-O2**.
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. Recompile or relink the application with **-qpdf2** and a minimum of **-O**.

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

Related information in the *XL C Compiler Reference*

-qpdf1, -qpdf2

-O, -qoptimize

Viewing profiling information with showpdf

To collect and view detailed information on function call and block statistics, compile with the **-qshowpdf** option and then use the **showpdf** utility. The following example shows how you can use profile-directed feedback (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 -O hello.c
   ```
2. Run the resulting executable program **a.out** using a typical data set or several typical data sets.
3. Run the **showpdf** utility to display the call and block counts for the executable file. If you used the **-qipa=pdfname** option during compilation, use the **-f** option to indicate the instrumentation file.

   ```
   showpdf -f instr1
   ```

The results will look similar to this:

```
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 information in the *XL C Compiler Reference*

-qpdf1, -qpdf2

-qshowpdf

Object level profile-directed feedback

In addition to optimizing entire executables, profile-directed feedback (PDF) can also be applied to specific objects. This can be an advantage in applications where patches or updates are distributed as object files or libraries rather than as executables. Also, specific areas of functionality in your application can be optimized without you needing to go through the process of relinking the entire application. In large applications, you can save the time and trouble that otherwise would have been spent relinking the application.

The process for using object level PDF is essentially the same as the standard PDF process but with a small change to the **-qpdf2** step. For object level PDF, compile your application using **-qpdf1**, execute the application with representative data, compile the application again with **-qpdf2** but now also use the **-qnoipa** option so that the linking step is skipped.

The steps below outline this process:

1. Compile your application using **-qpdf1**. For example:
   ```
   xlc -c -O3 -qpdf1 file1.c file2.c file3.c
   ```
   In this example, we are using the option **-O3** to indicate that we want a moderate level of optimization.
2. Link the object files to get an instrumented executable.
   ```
   xlc -O3 -qpdf1 file1.o file2.o file3.o
   ```
   Note: you must use the same optimization options. In this example, the optimization option **-O3**.
3. Run the instrumented executable with sample data that is representative of the data you want to optimize for.
   ```
   a.out < sample_data
   ```
4. Compile the application again using **-qpdf2**. Specify the **-qnoipa** option so that the linking step is skipped and PDF optimization is applied to the object files rather than to the entire executable. **Note**: you must use the same optimization options as in the previous steps. In this example, the optimization option **-O3**.
   ```
   xlc -c -O3 -qpdf2 -qnoipa file1.c file2.c file3.c
   ```
   The resulting output of this step are object files optimized for the sample data processed by the original instrumented executable. In this example, the optimized object files would be file1.o, file2.o, and file3.o. These can be linked using the system loader **ld** or by omitting the **-c** option in the **-qpdf2** step.

Notes:

- If you want to specify a file name for the profile that is created, use the **pdfname** suboption in both the **-qpdf1** and **-qpdf2** steps. For example:
  ```
  xlc -O3 -qpdf1=pdfname=myprofile file1.c file2.c file3.c
  ```
  Without the **pdfname** suboption, by default the file name will be ._pdf; the location of the file will be the current working directory or whatever directory you have set using the PDFDIR environment variable.
- You must use the same optimization options in each compilation and linking step.
- Because **-qnoipa** needs to be specified in the **-qpdf2** step so that linking of your object files is skipped, you will not be able to use interprocedural analysis (IPA) optimizations and object level PDF at the same time.

Other optimization options

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 Compiler Reference*.

Table 22. Selected compiler options for optimizing performance

Option	Description
-qignerrno	Allows the compiler to assume that `errno` 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 `sqrt`.)
-qsmallstack	Instructs the compiler to compact stack storage. Doing so may increase heap usage.
-qinline	Controls inlining by the low-level optimizer.
-qunroll	Independently controls loop unrolling. **-qunroll** is implicitly activated under -O3.
-qinlglue	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.
-qtbtable	Controls the generation of traceback table information.
-qnounwind	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.
-qnostrict	Allows the compiler to reorder 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). **-qnostrict** is used by default for optimization levels **-O3** and higher.
-qlargepage	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.

Related information in the *XL C Compiler Reference*

-qignerrno

-qsmallstack

-Q, -qinline

-qunroll / #pragma unroll

-qinlglue

-qtbtable

-qunwind

- -qstrict
- -qlargepage

Chapter 8. Debugging optimized code

Debugging optimized programs presents special usability problems. Optimization can change the sequence of operations, add or remove code, change variable data locations, and perform other transformations that make it difficult to associate the generated code with the original source statements.

For example:

Data location issues
: With an optimized program, it is not always certain where the most current value for a variable is located. For example, a value in memory may not be current if the most current value is being stored in a register. Most debuggers are incapable of following the removal of stores to a variable, and to the debugger it appears as though that variable is never updated, or possibly even set. This contrasts with no optimization where all values are flushed back to memory and debugging can be more effective and usable.

Instruction scheduling issues
: With an optimized program, the compiler may reorder instructions. That is, instructions may not be executed in the order the programmer would expect based on the sequence of lines in their original source code. Also, the sequence of instructions may not be contiguous. As the user steps through their program with a debugger, it may appear as if they are returning to a previously executed line in their code (interleaving of instructions).

Consolidating variable values
: Optimizations can result in the removal and consolidation of variables. For example, if a program has two expressions that assign the same value to two different variables, the compiler may substitute a single variable. This can inhibit debug usability because a variable that a programmer is expecting to see is no longer available in the optimized program.

There are a couple of different approaches you can take to improve debug capabilities while also optimizing your program:

Debug non-optimized code first
: Debug a non-optimized version of your program first, then recompile it with your desired optimization options. See "Debugging before optimization" on page 66 for some compiler options that are useful in this approach.

Use -qoptdebug
: When compiling with **-O3** optimization or higher, use the compiler option **-qoptdebug** to generate a pseudocode file that more accurately maps to how instructions and variable values will operate in an optimized program. With this option, when you load your program into a debugger, you will be debugging the pseudocode for the optimized program. See "Using -qoptdebug to help debug optimized programs" on page 67 for more information.

Understanding different results in optimized programs

Here are some reasons why an optimized program might produce different results from one that has not undergone the optimization process:

- Optimized code can fail if a program contains code that is not valid. The optimization process relies on your application conforming to language standards.
- If a program that works without optimization fails when you optimize, check the cross-reference listing and the execution flow of the program for variables that are used before they are initialized. Compile with the **-qinitauto=***hex_value* option to try to produce the incorrect results consistently. For example, using **-qinitauto=FF** gives variables an initial value of ″negative not a number″ (-NAN). Any operations on these variables will also result in NAN values. Other bit patterns (***hex_value***) may yield different results and provide further clues as to what is going on. Programs with uninitialized variables can appear to work properly when compiled without optimization, because of the default assumptions the compiler makes, but can fail when you optimize. Similarly, a program can appear to execute correctly after optimization, but fails at lower optimization levels or when run in a different environment.
- A variation on uninitialized storage. Referring to an automatic-storage variable by its address after the owning function has gone out of scope leads to a reference to a memory location that can be overwritten as other auto variables come into scope as new functions are called.

Use with caution debugging techniques that rely on examining values in storage. The compiler might have deleted or moved a common expression evaluation. It might have assigned some variables to registers, so that they do not appear in storage at all.

Debugging before optimization

First debug your program, then recompile it with your desired optimization options, and test the optimized program before placing the program into production. If the optimized code does not produce the expected results, you can attempt to isolate the specific optimization problems in a debugging session.

The following list presents options that provide specialized information, which can be helpful during the development of optimized code:

-qsmp=noopt
: If you are debugging SMP code, **-qsmp=noopt** ensures that the compiler performs only the minimum transformations necessary to parallelize your code and preserves maximum debug capability.

-qkeepparm
: Ensures that procedure parameters are stored on the stack even during optimization. This can negatively impact execution performance. The **-qkeepparm** option then provides access to the values of incoming parameters to tools, such as debuggers, simply by preserving those values on the stack.

-qlist
: Instructs the compiler to emit an object listing. The object listing includes hex and pseudo-assembly representations of the generated instructions, traceback tables, and text constants.

-qreport
: Instructs the compiler to produce a report of the loop transformations it

performed and how the program was parallelized. For **-qreport** to generate a listing, the options **-qhot** or **-qsmp** should also be specified.

-qinitauto
: Instructs the compiler to emit code that initializes all automatic variables to a given value.

-qextchk
: Generates additional symbolic information to allow the linker to do cross-file type checking of external variables and functions. This option requires the linker **-btypchk** option to be active.

-qipa=list
: Instructs the compiler to emit an object listing that provides information for IPA optimization.

You can also use the **snapshot** pragma to ensure to that certain variables are visible to the debugger at points in your application.

Using -qoptdebug to help debug optimized programs

The purpose of the **-qoptdebug** compiler option is to aid the debugging of optimized programs. It does this by creating pseudocode that maps more closely to the instructions and values of an optimized program than the original source code. When a program compiled with this option is loaded into a debugger, you will be debugging the pseudocode rather than your original source. By making optimizations explicit in pseudocode, you can gain a better understanding of how your program is really behaving under optimization. Files containing the pseudocode for your program will be generated with the file suffix `.optdbg`. Only line debugging is supported for this feature.

Compile your program as in the following example:

```
xlc myprogram.c -O3 -qhot -g -qoptdebug
```

In this example, your source file will be compiled to a.out. The pseudocode for the optimized program will be written to a file called myprogram.optdbg which can be referred to while debugging your program.

Notes:
- The **-g** or the **-qlinedebug** option must also be specified in order for the compiled executable to be debuggable. However, if neither of these options are specified, the pseudocode file `<output_file>.optdbg` containing the optimized pseudocode will still be generated.
- The **-qoptdebug** option only has an effect when one or more of the optimization options **-qhot**, **-qsmp**, **-qipa**, or **-qpdf** are specified, or when the optimization levels that imply these options are specified; that is, the optimization levels **-O3**, **-O4**, and **-O5**. The example shows the optimization options **-qhot** and **-O3**.

Debugging the optimized program

See the figures below as an aid to understanding how the compiler may apply optimizations to a simple program and how debugging it would differ from debugging your original source.

Figure 2 on page 68: Represents the original non-optimized code for a simple program. It presents a couple of optimization opportunities to the compiler. For example, the variables `z` and `d` are both assigned by the equivalent expressions `x +`

y. Therefore, these two variables can be consolidated in the optimized source. Also, the loop can be unrolled. In the optimized source, you would see iterations of the loop listed explicitly.

Figure 3: Represents a listing of the optimized source as shown in the debugger. Note the unrolled loop and the consolidation of values assigned by the x + y expression.

Figure 4 on page 69: Shows an example of stepping through the optimized source using the debugger. Note, there is no longer a correspondence between the line numbers for these statements in the optimized source as compared to the line numbers in the original source.

```
#include "stdio.h"

void foo(int x, int y, char* w)
{
 char* s = w+1;
 char* t = w+1;
 int z = x + y;
 int d = x + y;
 int a = printf("TEST\n");

 for (int i = 0; i < 4; i++)
  printf("%d %d %d %s %s\n", a, z, d, s, t);
 }

int main()
{
 char d[] = "DEBUG";
 foo(3, 4, d);
 return 0;
}
```

Figure 2. **Original code**

```
dbx> list
     1    3 |  void foo(long x, long y, char * w)
     2    9 |  {
     3           a = printf("TEST/n");
     4   12 |    @CSE0 = x + y;
     5           printf("%d %d %d %s %s/n",a,@CSE0,@CSE0,((char *)w  + 1),((char *)w  + 1));
     6           printf("%d %d %d %s %s/n",a,@CSE0,@CSE0,((char *)w  + 1),((char *)w  + 1));
     7           printf("%d %d %d %s %s/n",a,@CSE0,@CSE0,((char *)w  + 1),((char *)w  + 1));
     8           printf("%d %d %d %s %s/n",a,@CSE0,@CSE0,((char *)w  + 1),((char *)w  + 1));
     9   13 |    return;
    10         } /* function */
    11   15 |  long main()
    12   17 |  {
    13           d$init$0 = "DEBUG";
    14   18 |    foo(3,4,&d)
    15   19 |    rstr = 0;
    16           return rstr;
    17   20 |  } /* function */
```

Figure 3. **dbx debugger listing**

```
dbx> stop at 3
[1] stop at "myprogram.o.optdbg":3
dbx> run
TEST
[1] stopped in foo(int,int,char*) at line 3 in file "myprogram.o.optdbg" ($t1)
    3       16 |    @CSE0 = x + y;
dbx> step
stopped in foo(int,int,char*) at line 4 in file "myprogram.o.optdbg" ($t1)
    4               printf("%d %d %d %s %s/n",a,@CSE0,@CSE0,((char *)w  + 1),((char *)w  + 1));
dbx> step
3 7 7 EBUG EBUG
stopped in foo(int,int,char*) at line 5 in file "myprogram.o.optdbg" ($t1)
    5               printf("%d %d %d %s %s/n",a,@CSE0,@CSE0,((char *)w  + 1),((char *)w  + 1));
dbx> cont
3 7 7 EBUG EBUG
3 7 7 EBUG EBUG
3 7 7 EBUG EBUG

execution completed
```

Figure 4. **Stepping through optimized source**

Chapter 9. Coding your application to improve performance

Chapter 7, "Optimizing your applications," on page 45 discusses the various compiler options that XL 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 programming techniques that can improve performance of your code:

- "Find faster input/output techniques"
- "Reduce function-call overhead"
- "Manage memory efficiently" on page 73
- "Optimize variables" on page 73
- "Manipulate strings efficiently" on page 74
- "Optimize expressions and program logic" on page 74
- "Optimize operations in 64-bit mode" on page 75

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 expected_value** preprocessor directive so that the compiler can optimize for common values used with a function.
- 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 function as static whenever possible. This can speed up calls to the function.
- 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. 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 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.

Related information in the *XL C Compiler Reference*

#pragma expected_value

-qisolated_call / #pragma isolated_call

#pragma disjoint

-qipa

Manage memory efficiently

- In a structure, declare the largest members first.
- In a structure, place variables near each other if they are frequently used together.
- 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.
- C++ 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 runtime 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 runtime 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 information in the *XL C Compiler Reference*

-qisolated_call / #pragma isolated_call

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 information in the *XL C Compiler Reference*

-qro / #pragma strings

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.
- 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.
- 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.

Chapter 10. Using the high performance libraries

IBM XL C for AIX, V10.1 is shipped with a set of libraries for high-performance mathematical computing:

- The Mathematical Acceleration Subsystem (MASS) is a set of libraries of tuned mathematical intrinsic functions that provide improved performance over the corresponding standard system math library functions. MASS is described in "Using the Mathematical Acceleration Subsystem libraries (MASS)."
- The Basic Linear Algebra Subprograms (BLAS) are a set of routines which provide matrix/vector multiplication functions tuned for PowerPC architectures. The BLAS functions are described in "Using the Basic Linear Algebra Subprograms – BLAS" on page 85.

Using the Mathematical Acceleration Subsystem libraries (MASS)

The MASS libraries consist of a library of scalar described in "Using the scalar library"; and a set of vector libraries tuned for specific architectures, described in "Using the vector libraries" on page 79. The functions contained in both scalar and vector libraries are automatically called at certain levels of optimization, but you can also call them explicitly in your programs. Note that the accuracy and exception handling might not be identical in MASS functions and system library functions.

"Compiling and linking a program with MASS" on page 85 describes how to compile and link a program that uses the MASS libraries, and how to selectively use the MASS scalar library functions in conjunction with the regular system libraries.

Related external information

Mathematical Acceleration Subsystem Web site, available at http://www.ibm.com/software/awdtools/mass/

Using the scalar library

The MASS scalar library libmass.a contains an accelerated set of frequently used math intrinsic functions that provide improved performance over the corresponding standard system library functions. The MASS scalar functions are used when explicitly linking libmass.a , but are also available automatically when you compile programs with any of the following options:

- **-qhot -O3**
- **-O4**
- **-O5**

With these options, the compiler automatically uses the faster MASS functions for most math library functions. In fact, the compiler first tries to ″vectorize″ calls to math library functions by replacing them with the equivalent MASS vector functions; if it cannot do so, it uses the MASS scalar functions. When the compiler performs this automatic replacement of math library functions, it uses versions of the MASS functions contained in the system library `libxlopt.a`. You do not need to add any special calls to the MASS functions in your code, or to link to the `libxlopt` library.

If you are not using any of the optimization options listed above, and want to explicitly call the MASS scalar functions, you can do so as follows:

The MASS scalar functions accept double-precision parameters and return a double-precision result, or accept single-precision parameters and return a single-precision result, except `sincos` which gives 2 double-precision results. They are summarized in Table 23.

Table 23. MASS scalar functions

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
acos	acosf	Returns the arccosine of `x`	double acos (double x);	float acosf (float x);
acosh	acoshf	Returns the hyperbolic arccosine of `x`	double acosh (double x);	float acoshf (float x);
	anint	Returns the rounded integer value of `x`		float anint (float x);
asin	asinf	Returns the arcsine of `x`	double asin (double x);	float asinf (float x);
asinh	asinhf	Returns the hyperbolic arcsine of `x`	double asinh (double x);	float asinhf (float x);
atan2	atan2f	Returns the arctangent of `x/y`	double atan2 (double x, double y);	float atan2f (float x, float y);
atan	atanf	Returns the arctangent of `x`	double atan (double x);	float atanf (float x);
atanh	atanhf	Returns the hyperbolic arctangent of `x`	double atanh (double x);	float atanhf (float x);
cbrt	cbrtf	Returns the cube root of `x`	double cbrt (double x);	float cbrtf (float x);
copysign	copysignf	Returns `x` with the sign of `y`	double copysign (double x,double y);	float copysignf (float x);
cos	cosf	Returns the cosine of `x`	double cos (double x);	float cosf (float x);
cosh	coshf	Returns the hyperbolic cosine of `x`	double cosh (double x);	float coshf (float x);
cosisin		Returns a complex number with the real part the cosine of `x` and the imaginary part the sine of `x`.	double_Complex cosisin (double);	
dnint		Returns the nearest integer to `x` (as a double)	double dnint (double x);	
erf	erff	Returns the error function of `x`	double erf (double x);	float erff (float x);
erfc	erfcf	Returns the complementary error function of `x`	double erfc (double x);	float erfcf (float x);
exp	expf	Returns the exponential function of `x`	double exp (double x);	float expf (float x);
expm1	expm1f	Returns (the exponential function of `x`) - 1	double expm1 (double x);	float expm1f (float x);

Table 23. MASS scalar functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
hypot	hypotf	Returns the square root of $x^2 + y^2$	double hypot (double x, double y);	float hypotf (float x, float y);
lgamma	lgammaf	Returns the natural logarithm of the absolute value of the Gamma function of `x`	double lgamma (double x);	float lgammaf (float x);
log	logf	Returns the natural logarithm of `x`	double log (double x);	float logf (float x);
log10	log10f	Returns the base 10 logarithm of `x`	double log10 (double x);	float log10f (float x);
log1p	log1pf	Returns the natural logarithm of (`x` + 1)	double log1p (double x);	float log1pf (float x);
rsqrt		Returns the reciprocal of the square root of `x`	double rsqrt (double x);	
sin	sinf	Returns the sine of `x`	double sin (double x);	float sinf (float x);
sincos		Sets `*s` to the sine of `x` and `*c` to the cosine of `x`	void sincos (double x, double* s, double* c);	
sinh	sinhf	Returns the hyperbolic sine of `x`	double sinh (double x);	float sinhf (float x);
sqrt		Returns the square root of `x`	double sqrt (double x);	
tan	tanf	Returns the tangent of `x`	double tan (double x);	float tanf (float x);
tanh	tanhf	Returns the hyperbolic tangent of `x`	double tanh (double x);	float tanhf (float x);

Notes:

- The trigonometric functions (`sin`, `cos`, `tan`) return NaN (Not-a-Number) for large arguments (where the absolute value is greater than 2^{50}pi).
- In some cases, the MASS functions are not as accurate as the `libm.a` library, and they might handle edge cases differently (`sqrt(Inf)`, for example).
- See the *Mathematical Acceleration Subsystem Web site* for accuracy comparisons with `libm.a`.

Related external information

Mathematical Acceleration Subsystem Web site, available at http://www.ibm.com/software/awdtools/mass/

Using the vector libraries

When you compile programs with any of the following options:

- **-qhot -O3**
- **-O4**
- **-O5**

the compiler automatically attempts to vectorize calls to system math functions by calling the equivalent MASS vector functions (with the exceptions of functions `vdnint`, `vdint`, `vsincos`, `vssincos`, `vcosisin`, `vscosisin`, `vqdrt`, `vsqdrt`, `vrqdrt`, `vsrqdrt`, `vpopcnt4`, and `vpopcnt8`). For automatic vectorization, the compiler uses versions of the MASS functions contained in the system library `libxlopt.a`. You do not need to add any special calls to the MASS functions in your code, or to link to the libxlopt library.

If you are not using any of the optimization options listed above, and want to explicitly call any of the MASS vector functions, you can do so by including the file in your source files and linking your application with the appropriate vector library. (Information on linking is provided in "Compiling and linking a program with MASS" on page 85.)

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 a PPC970 machine, this library is the recommended choice.

libmassvp5.a
Contains some functions that have been tuned for the POWER5 architecture. The remaining functions are identical to those in libmassv.a.

libmassvp6.a
Contains some functions that have been tuned for the POWER6 architecture. The remaining functions are identical to those in libmassv.a.

All libraries can be used in either 32-bit or 64-bit mode.

The single-precision and double-precision floating-point functions contained in the vector libraries are summarized in . The integer functions contained in the vector libraries are summarized in Table 25 on page 83.

With the exception of a few functions (described below), all of the floating-point functions in the vector libraries accept three parameters:

- A double-precision (for double-precision functions) or single-precision (for single-precision functions) vector output parameter
- A double-precision (for double-precision functions) or single-precision (for single-precision functions) vector input parameter
- An integer vector-length parameter.

The functions are of the form

```
function_name (y,x,n)
```

where *y* is the target vector, *x* is the source 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 examples, the following code: outputs a vector *y* of length 500 whose elements are .

The functions `vdiv`, `vsincos`, `vpow`, and `vatan2` (and their single-precision versions, `vsdiv`, `vssincos`, `vspow`, and `vsatan2`) take four parameters. The functions `vdiv`, `vpow`, 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 `vpow` 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)` and `vscosisin(y,x,n)`, *x* is a vector of *n* elements and the function outputs a vector *y* of *n* `complex` elements of the form (cos(x[i]),sin(x[i])). If **-D__nocomplex** is used (see note in Table 24), the output vector holds y[0][i] = cos(x[i]) and y[1][i] = sin(x[i]), where i=0,..,*n-1.

Table 24. MASS floating-point vector functions

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
vacos	vsacos	Sets `y[i]` to the arc cosine of `x[i]`, for i=0,..,*n–1	void vacos (double y[], double x[], int *n);	void vsacos (float y[], float x[], int *n);
vacosh	vsacosh	Sets `y[i]` to the hyperbolic arc cosine of `x[i]`, for i=0,..,*n–1	void vacosh (double y[], double x[], int *n);	void vsacosh (float y[], float x[], int *n);
vasin	vsasin	Sets `y[i]` to the arc sine of `x[i]`, for i=0,..,*n–1	void vasin (double y[], double x[], int *n);	void vsasin (float y[], float x[], int *n);
vasinh	vsasinh	Sets `y[i]` to the hyperbolic arc sine of `x[i]`, for i=0,..,*n–1	void vasinh (double y[], double x[], int *n);	void vsasinh (float y[], float x[], int *n);
vatan2	vsatan2	Sets `z[i]` to the arc tangent of `x[i]/y[i]`, for i=0,..,*n–1	void vatan2 (double z[], double x[], double y[], int *n);	void vsatan2 (float z[], float x[], float y[], int *n);
vatanh	vsatanh	Sets `y[i]` to the hyperbolic arc tangent of `x[i]`, for i=0,..,*n–1	void vatanh (double y[], double x[], int *n);	void vsatanh (float y[], float x[], int *n);
vcbrt	vscbrt	Sets `y[i]` to the cube root of `x[i]`, for i=0,..,*n-1	void vcbrt (double y[], double x[], int *n);	void vscbrt (float y[], float x[], int *n);
vcos	vscos	Sets `y[i]` to the cosine of `x[i]`, for i=0,..,*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,..,*n–1	void vcosh (double y[], double x[], int *n);	void vscosh (float y[], float x[], int *n);
vcosisin[1]	vscosisin[1]	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,..,*n–1	void vcosisin (double _Complex y[], double x[], int *n);	void vscosisin (float _Complex y[], float x[], int *n);

Table 24. MASS floating-point vector functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
vdint		Sets y[i] to the integer truncation of x[i], for i=0,..,*n–1	void vdint (double y[], double x[], int *n);	
vdiv	vsdiv	Sets z[i] to x[i]/y[i], for i=0,..,*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,..,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,..,*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,..,*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,..,*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,..,*n–1	void vlog10 (double y[], double x[], int *n);	void vslog10 (float y[], float x[], int *n);
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);
vqdrt	vsqdrt	Sets y[i] to the fourth root of x[i], for i=0,..,*n-1	void vqdrt (double y[], double x[], int *n);	void vsqdrt (float y[], float x[], int *n);
vrcbrt	vsrcbrt	Sets y[i] to the reciprocal of the cube root of x[i], for i=0,..,*n-1	void vrcbrt (double y[], double x[], int *n);	void vsrcbrt (float y[], float x[], 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);
vrqdrt	vsrqdrt	Sets y[i] to the reciprocal of the fourth root of x[i], for i=0,..,*n-1	void vrqdrt (double y[], double x[], int *n);	void vsrqdrt (float y[], float x[], int *n);

Table 24. MASS floating-point vector functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
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);

Note:

1. By default, these functions use the `__Complex` 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 **-D__nocomplex**. 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);`

Integer functions are of the form *function_name* (*x*[], **n*), where `x[]` is a vector of 4-byte (for `vpopcnt4`) or 8-byte (for `vpopcnt8`) numeric objects (integral or floating-point), and `*n` is the vector length.

Table 25. MASS integer vector library functions

Function	Description	Prototype
vpopcnt4	Returns the total number of 1 bits in the concatenation of the binary representation of `x[i]`, for i=0,..,*n–1 , where `x` is a vector of 32-bit objects.	unsigned int vpopcnt4 (void *x, int *n)
vpopcnt8	Returns the total number of 1 bits in the concatenation of the binary representation of `x[i]`, for i=0,..,*n–1 , where `x` is a vector of 64-bit objects.	unsigned int vpopcnt8 (void *x, int *n)

Overlap of input and output vectors

In most applications, the MASS vector functions are called with disjoint input and output vectors; that is, the two vectors do not overlap in memory. Another common usage scenario is to call them with the same vector for both input and output parameters (for example, `vsin (y, y, &n)`). For other kinds of overlap, be sure to observe the following restrictions, to ensure correct operation of your application:

- For calls to vector functions that take one input and one output vector (for example, `vsin (y, x, &n)`):

  The vectors `x[0:n-1]` and `y[0:n-1]` must be either disjoint or identical, or the address of `x[0]` must be greater than the address of `y[0]`. That is, if `x` and `y` are not the same vector, the address of `y[0]` must not fall within the range of addresses spanned by `x[0:n-1]`, or unexpected results may be obtained.
- For calls to vector functions that take two input vectors (for example, `vatan2 (y, x1, x2, &n)`):

  The previous restriction applies to both pairs of vectors `y`,`x1` and `y`,`x2`. That is, if `y` is not the same vector as `x1`, the address of `y[0]` must not fall within the range of addresses spanned by `x1[0:n-1]`; if `y` is not the same vector as `x2`, the address of `y[0]` must not fall within the range of addresses spanned by `x2[0:n-1]`.
- For calls to vector functions that take two output vectors (for example, `vsincos (y1, y2, x, &n)`):

  The above restriction applies to both pairs of vectors `y1`,`x` and `y2`,`x`. That is, if `y1` and `x` are not the same vector, the address of `y1[0]` must not fall within the range of addresses spanned by `x[0:n-1]`; if `y2` and `x` are not the same vector, the address of `y2[0]` must not fall within the range of addresses spanned by `x[0:n-1]`. Also, the vectors `y1[0:n-1]` and `y2[0:n-1]` must be disjoint.

Consistency of MASS vector functions

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

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.

The following functions are consistent in all versions of the library: `vcbrt`, `vscbrt`, `vrcbrt`, `vsrcbrt`, `vlog`, `vsin`, `vssin`, `vcos`, `vscos`, `vsexp`, `vacos`, `vasin`, `vrqdrt`, `vsqdrt`, `vsrqdrt`, `vacosh`, `vsacosh`, `vasinh`, `vsasinh`, `vtanh`, `vstanh`. The following functions are consistent in `libmassvp3.a`, `libmassvp4.a`, `libmassvp5.a`, and `libmassvp6.a`: `vsqrt`, `vrsqrt`. The following functions are consistent in `libmassvp4.a`, `libmassvp5.a`, and `libmassvp6.a`: `vrec`, `vsrec`, `vdiv`, `vsdiv`, `vexp`. The following function is consistent in `libmassv.a`, `libmassvp5.a`, and `libmassvp6.a`: `vsrsqrt`. Older, inconsistent versions of some of these functions are available on the *Mathematical Acceleration Subsystem for AIX Web site*. If consistency is not required, there may be a performance advantage to using the older versions. For more information on consistency and avoiding inconsistency with the vector libraries, as well as performance and accuracy data, see the *Mathematical Acceleration Subsystem Web site*.

Related information in the *XL C Compiler Reference*

-D

Related external information

Mathematical Acceleration Subsystem for AIX Web site, available at http://www.ibm.com/software/awdtools/mass/aix

Mathematical Acceleration Subsystem Web site, available at http://www.ibm.com/software/awdtools/mass/

Compiling and linking a program with MASS

To compile an application that calls the functions in the MASS libraries, specify **mass** and **massv** (or **massvp3**, **massvp4**, **massvp5**, or **massvp6**) on the **-l** linker option.

For example, if the MASS libraries are installed in the default directory, you could specify:

```
xlc progc.c -o progc -lmass -lmassv
```

The MASS functions must run in the default rounding mode and floating-point exception trapping settings.

Using libmass.a with the math system library

If you wish to use the libmass.a scalar library for some functions and the normal math library 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 fasttan.exp with the following line:
   ```
   tan
   ```
2. Create a shared object from the export list with the **ld** command, linking with the libmass.a library. For example:
   ```
   ld -bexport:fasttan.exp -o fasttan.o -bnoentry -lmass -bmodtype:SRE
   ```
3. Archive the shared object into a library with the **ar** command. For example:
   ```
   ar -q libfasttan.a fasttan.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 math library. For example:
   ```
   xlc sample.c -o sample -Ldir_containing_libfasttan -lfasttan -lm
   ```

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

Related external information

- **ar** and **ld** in the *AIX Commands Reference, Volumes 1 - 6*

Using the Basic Linear Algebra Subprograms – BLAS

Four Basic Linear Algebra Subprograms (BLAS) functions are shipped with XL C in the `libxlopt` library. The functions consist of the following:

- `sgemv` (single-precision) and `dgemv` (double-precision), which compute the matrix-vector product for a general matrix or its transpose
- `sgemm` (single-precision) and `dgemm` (double-precision), which perform combined matrix multiplication and addition for general matrices or their transposes

Because the BLAS routines are written in Fortran, all parameters are passed to them by reference, and all arrays are stored in column-major order.

Note: Some error-handling code has been removed from the BLAS functions in `libxlopt`, and no error messages are emitted for calls to the these functions.

"BLAS function syntax" describes the prototypes and parameters for the XL C BLAS functions. The interfaces for these functions are similar to those of the equivalent BLAS functions shipped in IBM's Engineering and Scientific Subroutine Library (ESSL); for more detailed information and examples of usage of these functions, you may wish to consult the *Engineering and Scientific Subroutine Library Guide and Reference*, available at http://publib.boulder.ibm.com/clresctr/windows/public/esslbooks.html.

"Linking the libxlopt library" on page 88 describes how to link to the XL C `libxlopt` library if you are also using a third-party BLAS library.

BLAS function syntax

The prototypes for the `sgemv` and `dgemv` functions are as follows:

```
void sgemv(const char *trans, int *m, int *n, float *alpha,
     void *a, int *lda, void *x, int *incx,
     float *beta, void *y, int *incy);
```

```
void dgemv(const char *trans, int *m, int *n, double *alpha,
     void *a, int *lda, void *x, int *incx,
      double *beta, void *y, int *incy);
```

The parameters are as follows:

trans
: is a single character indicating the form of the input matrix *a*, where:
  - `'N'` or `'n'` indicates that *a* is to be used in the computation
  - `'T'` or `'t'` indicates that the transpose of *a* is to be used in the computation

m
: represents:
  - the number of rows in input matrix *a*
  - the length of vector *y*, if `'N'` or `'n'` is used for the *trans* parameter
  - the length of vector *x*, if `'T'` or `'t'` is used for the *trans* parameter

  The number of rows must be greater than or equal to zero, and less than the leading dimension of the matrix *a* (specified in *lda*)

n
: represents:
  - the number of columns in input matrix *a*
  - the length of vector *x*, if `'N'` or `'n'` is used for the *trans* parameter
  - the length of vector *y*, if `'T'` or `'t'` is used for the *trans* parameter

  The number of columns must be greater than or equal to zero.

alpha
: is the scaling constant for matrix *a*

a
: is the input matrix of `float` (for `sgemv`) or `double` (for `dgemv`) values

lda is the leading dimension of the array specified by *a*. The leading dimension must be greater than zero. The leading dimension must be greater than or equal to 1 and greater than or equal to the value specified in *m*.

x is the input vector of `float` (for `sgemv`) or `double` (for `dgemv`) values.

incx
is the stride for vector *x*. It can have any value.

beta
is the scaling constant for vector *y*

y is the output vector of `float` (for `sgemv`) or `double` (for `dgemv`) values.

incy
is the stride for vector *y*. It must not be zero.

Note: Vector *y* must have no common elements with matrix *a* or vector *x*; otherwise, the results are unpredictable.

The prototypes for the `sgemm` and `dgemm` functions are as follows:

```
void sgemm(const char *transa, const char *transb,
   int *l, int *n, int *m, float *alpha,
   const void *a, int *lda, void *b, int *ldb,
   float *beta, void *c, int *ldc);
void dgemm(const char *transa, const char *transb,
   int *l, int *n, int *m, double *alpha,
   const void *a, int *lda, void *b, int *ldb,
   double *beta, void *c, int *ldc);
```

The parameters are as follows:

transa
is a single character indicating the form of the input matrix *a*, where:
- `'N'` or `'n'` indicates that *a* is to be used in the computation
- `'T'` or `'t'` indicates that the transpose of *a* is to be used in the computation

transb
is a single character indicating the form of the input matrix *b*, where:
- `'N'` or `'n'` indicates that *b* is to be used in the computation
- `'T'` or `'t'` indicates that the transpose of *b* is to be used in the computation

l represents the number of rows in output matrix *c*. The number of rows must be greater than or equal to zero, and less than the leading dimension of *c*.

n represents the number of columns in output matrix *c*. The number of columns must be greater than or equal to zero.

m represents:
- the number of columns in matrix *a*, if `'N'` or `'n'` is used for the *transa* parameter
- the number of rows in matrix *a*, if `'T'` or `'t'` is used for the *transa* parameter

and:
- the number of rows in matrix *b*, if `'N'` or `'n'` is used for the *transb* parameter
- the number of columns in matrix *b*, if `'T'` or `'t'` is used for the *transb* parameter

m must be greater than or equal to zero.

alpha
: is the scaling constant for matrix *a*

a is the input matrix *a* of `float` (for `sgemm`) or `double` (for `dgemm`) values

lda is the leading dimension of the array specified by *a*. The leading dimension must be greater than zero. If *transa* is specified as `'N'` or `'n'`, the leading dimension must be greater than or equal to 1. If *transa* is specified as `'T'` or `'t'`, the leading dimension must be greater than or equal to the value specified in *m*.

b is the input matrix *b* of `float` (for `sgemm`) or `double` (for `dgemm`) values.

ldb is the leading dimension of the array specified by *b*. The leading dimension must be greater than zero. If *transb* is specified as `'N'` or `'n'`, the leading dimension must be greater than or equal to the value specified in *m*. If *transa* is specified as `'T'` or `'t'`, the leading dimension must be greater than or equal to the value specified in *n*.

beta
: is the scaling constant for matrix *c*

c is the output matrix *c* of `float` (for `sgemm`) or `double` (for `dgemm`) values.

ldc is the leading dimension of the array specified by *c*. The leading dimension must be greater than zero. If *transb* is specified as `'N'` or `'n'`, the leading dimension must be greater than or equal to 0 and greater than or equal to the value specified in *l*.

Note: Matrix *c* must have no common elements with matrices *a* or *b*; otherwise, the results are unpredictable.

Linking the libxlopt library

By default, the `libxlopt` library is linked with any application you compile with XL C. However, if you are using a third-party BLAS library, but want to use the BLAS routines shipped with `libxlopt`, you must specify the `libxlopt` library before any other BLAS library on the command line at link time. For example, if your other BLAS library is called `libblas.a`, you would compile your code with the following command:

```
xlc app.c -lxlopt -lblas
```

The compiler will call the `sgemv`, `dgemv`, `sgemm`, and `dgemm` functions from the `libxlopt` library, and all other BLAS functions in the `libblas.a` library.

Chapter 11. Parallelizing your programs

The compiler offers you three methods of implementing shared memory program parallelization. These are:

- Automatic parallelization of countable program loops, which are defined in "Countable loops" on page 90. An overview of the compiler's automatic parallelization capabilities is provided in "Enabling automatic parallelization" on page 91.
- Explicit parallelization of countable loops using IBM SMP directives. An overview of the IBM SMP directives is provided in "Using IBM SMP directives" on page 91.
- Explicit parallelization of C program code using pragma directives compliant to the OpenMP Application Program Interface specification. An overview of the OpenMP directives is provided in "Using OpenMP directives" on page 92.

All methods of program parallelization are enabled when the **-qsmp** compiler option is in effect without the **omp** suboption. You can enable strict OpenMP compliance with the **-qsmp=omp** compiler option, but doing so will disable automatic parallelization.

Note: The **-qsmp** option must only be used together with thread-safe compiler invocation modes (those that contain the **_r** suffix).

Parallel regions of program code are executed by multiple threads, possibly running on multiple processors. The number of threads created is determined by environment variables and calls to library functions. Work is distributed among available threads according to scheduling algorithms specified by the environment variables. For any of the methods of parallelization, you can use the XLSMPOPTS environment variable and its suboptions to control thread scheduling; for more information on this environment variable, see *XLSMPOPTS* in the *XL C Compiler Reference*. If you are using OpenMP constructs, you can use the OpenMP environment variables to control thread scheduling; for information on OpenMP environment variables, see *OpenMP environment variables for parallel processing* in the *XL C Compiler Reference*. For more information on both IBM SMP and OpenMP built-in functions, see *Built-in functions for parallel processing* in the *XL C Compiler Reference*.

For a complete discussion on how threads are created and utilized, refer to the *OpenMP Application Program Interface Language Specification*, available at http://www.openmp.org.

Related information

"Using shared-memory parallelism (SMP)" on page 55
Many 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**.

Related information in the *XL C Compiler Reference*

XLSMPOPTS

OpenMP environment variables for parallel processing

Built-in functions for parallel processing

Related external information

OpenMP Application Program Interface Language Specification, available at http://www.openmp.org

Countable loops

Loops are considered to be countable if they take any of the following forms:

Countable for loop syntax with single statement

```
>>--for--(--+--------------------+--;--exit_condition--;--increment_expression--)------->
            '-iteration_variable-'

>--statement------------------------------------------------------------------------><
```

Countable for loop syntax with statement block

```
>>--for--(--+--------------------+--;--+------------+--)-------------------------------->
            '-iteration_variable-'     '-expression-'

>--{--+------------------+--+----------------+--increment_expression--+----------------+--}------><
      '-declaration_list-'  '-statement_list-'                        '-statement_list-'
```

Countable while loop syntax

```
>>--while--(--exit_condition--)--------------------------------------------------------->

>--{--+------------------+--+----------------+--increment_expression--}----------------><
      '-declaration_list-'  '-statement_list-'
```

Countable do while loop syntax

```
>>--do--{--+------------------+--+----------------+--increment_expression--}--while--(--exit_condition--)------><
           '-declaration_list-'  '-statement_list-'
```

The following definitions apply to the above syntax diagrams:

iteration_variable
: is a signed integer that has either automatic or register storage class, does not have its address taken, and is not modified anywhere in the loop except in the *increment_expression*.

exit_condition
: takes the following form:

```
|--increment_variable--+-<=-+--expression-------------------------------|
                       +-<--+
                       +->=-+
                       '->--'
```

where *expression* is a loop-invariant signed integer expression. *expression* cannot reference external or static variables, pointers or pointer expressions, function calls, or variables that have their address taken.

increment_expression
: takes any of the following forms:

- ++*iteration_variable*

- *--iteration_variable*
- *iteration_variable++*
- *iteration_variable--*
- *iteration_variable += increment*
- *iteration_variable -= increment*
- *iteration_variable = iteration_variable + increment*
- *iteration_variable = increment + iteration_variable*
- *iteration_variable = iteration_variable - increment*

where *increment* is a loop-invariant signed integer expression. The value of the expression is known at run time and is not 0. *increment* cannot reference external or static variables, pointers or pointer expressions, function calls, or variables that have their address taken.

Enabling automatic parallelization

The compiler can automatically locate and where possible parallelize all countable loops in your program code. A loop is considered to be countable if it has any of the forms shown in "Countable loops" on page 90, and:
- There is no branching into or out of the loop.
- The increment expression is not within a critical section.

In general, a countable loop is automatically parallelized only if all of the following conditions are met:
- The order in which loop iterations start or end does not affect the results of the program.
- The loop does not contain I/O operations.
- Floating point reductions inside the loop are not affected by round-off error, unless the **-qnostrict** option is in effect.
- The **-qnostrict_induction** compiler option is in effect.
- The **-qsmp=auto** compiler option is in effect.
- The compiler is invoked with a thread-safe compiler mode.

Using IBM SMP directives

IBM SMP directives exploit shared memory parallelism through the parallelization of countable loops. A loop is considered to be countable if it has any of the forms described in "Countable loops" on page 90. The XL C compiler provides pragma directives that you can use to improve on automatic parallelization performed by the compiler. Pragmas fall into two general categories:
1. Pragmas that give you explicit control over parallelization. Use these pragmas to force or suppress parallelization of a loop (**#pragma ibm parallel_loop** and **#pragma ibm sequential_loop**), apply specific parallelization algorithms to a loop (**#pragma ibm schedule**), and synchronize access to shared variables using critical sections (**#pragma ibm critical**).
2. Pragmas that let you give the compiler information on the characteristics of a specific countable loop (**#pragma ibm independent_calls**, **#pragma ibm independent_loop**, **#pragma ibm iterations**, **#pragma ibm permutation**). The compiler uses this information to perform more efficient automatic parallelization of the loop.

IBM SMP directive syntax

```
►►──#pragma ibm──pragma_name_and_args──countable_loop──────────────────────────►◄
```

Pragma directives must appear immediately before the countable loop to which they apply. More than one parallel processing pragma directive can be applied to a countable loop. For example:

```
#pragma ibm independent_loop
#pragma ibm independent_calls
#pragma ibm schedule(static,5)
countable_loop
```

Some pragma directives are mutually exclusive of each other, such as, for example, the parallel_loop sequential_loop directives. If mutually exclusive pragmas are specified for the same loop, the pragma last specified applies to the loop.

Other pragmas, if specified repeatedly for a given loop, have an additive effect. For example:

```
#pragma ibm permutation (a,b)
#pragma ibm permutation (c)
```

is equivalent to:

```
#pragma ibm permutation
(a,b,c)
```

For a pragma-by-pragma description of the IBM SMP directives, refer to *Pragma directives for parallel processing* in the *XL C Compiler Reference*.

Related information in the *XL C Compiler Reference*

Pragma directives for parallel processing

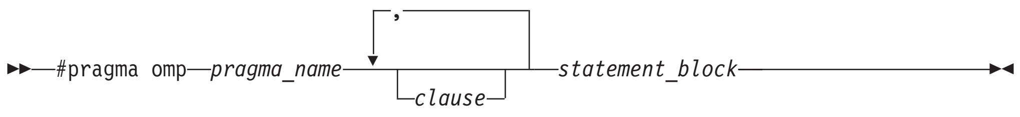
Using OpenMP directives

OpenMP directives exploit shared memory parallelism by defining various types of parallel regions. Parallel regions can include both iterative and non-iterative segments of program code.

Pragmas fall into five general categories:

1. Pragmas that let you define parallel regions in which work is done by threads in parallel (**#pragma omp parallel**). Most of the OpenMP directives either statically or dynamically bind to an enclosing parallel region.
2. Pragmas that let you define how work will be distributed or shared across the threads in a parallel region (**#pragma omp section**, **#pragma omp ordered**, **#pragma omp single**, **#pragma omp task**).
3. Pragmas that let you control synchronization among threads (**#pragma omp atomic**, **#pragma omp master**, **#pragma omp barrier**, **#pragma omp critical**, **#pragma omp flush**).
4. Pragmas that let you define the scope of data visibility across threads (**#pragma omp threadprivate**).
5. Pragmas for task synchronization (**#pragma omp taskwait**, **#pragma omp barrier**)

OpenMP directive syntax



Pragma directives generally appear immediately before the section of code to which they apply. For example, the following example defines a parallel region in which iterations of a for loop can run in parallel:

```
#pragma omp parallel
{
  #pragma omp for
    for (i=0; i<n; i++)
      ...
}
```

This example defines a parallel region in which two or more non-iterative sections of program code can run in parallel:

```
#pragma omp sections
{
  #pragma omp section
     structured_block_1
          ...
  #pragma omp section
     structured_block_2
          ...
      ....
}
```

For a pragma-by-pragma description of the IBM SMP directives, refer to *Pragma directives for parallel processing* in the *XL C Compiler Reference*.

Related information in the *XL C Compiler Reference*

Pragma directives for parallel processing

OpenMP built-in functions

OpenMP environment variables for parallel processing

Shared and private variables in a parallel environment

Variables can have either shared or private context in a parallel environment. Variables in shared context are visible to all threads running in associated parallel loops. Variables in private context are hidden from other threads. Each thread has its own private copy of the variable, and modifications made by a thread to its copy are not visible to other threads.

The default context of a variable is determined by the following rules:

- Variables with `static` storage duration are shared.
- Dynamically allocated objects are shared.
- Variables with automatic storage duration are private.
- Variables in heap allocated memory are shared. There can be only one shared heap.
- All variables defined outside a parallel construct become shared when the parallel loop is encountered.

- Loop iteration variables are private within their loops. The value of the iteration variable after the loop is the same as if the loop were run sequentially.
- Memory allocated within a parallel loop by the `alloca` function persists only for the duration of one iteration of that loop, and is private for each thread.

The following code segments show examples of these default rules:

```
int E1;                        /* shared static     */

void main (argvc,...) {        /* argvc is shared   */
   int i;                       /* shared automatic  */

void *p = malloc(...);       /* memory allocated by malloc   */
                                /* is accessible by all threads */
                                /* and cannot be privatized     */

#pragma omp parallel firstprivate (p)
   {
     int b;                     /* private automatic  */
     static int s;              /* shared static      */

     #pragma omp for
     for (i =0;...) {
       b = 1;                   /* b is still private here !    */
       foo (i);                 /* i is private here because it */
                                /* is an iteration variable     */
      }


#pragma omp parallel
     {
       b = 1;                   /* b is shared here because it  */
                                /* is another parallel region   */
     }
   }
 }


int E2;                        /*shared static */

void foo (int x) {             /* x is private for the parallel */
                                /* region it was called from     */

int c;                       /* the same */
 ... }
```

The compiler can privatize some shared variables if it is possible to do so without changing the semantics of the program. For example, if each loop iteration uses a unique value of a shared variable, that variable can be privatized. Privatized shared variables are reported by the **-qinfo=private** option. Use critical sections to synchronize access to all shared variables not listed in this report.

Some OpenMP preprocessor directives let you specify visibility context for selected data variables. A brief summary of data scope attribute clauses are listed below:

Data scope attribute clause	**Description**
private	The **private** clause declares the variables in the list to be private to each thread in a team.
firstprivate	The **firstprivate** clause provides a superset of the functionality provided by the private clause.

Data scope attribute clause	Description
lastprivate	The **lastprivate** clause provides a superset of the functionality provided by the private clause.
shared	The **shared** clause shares variables that appear in the list among all the threads in a team. All threads within a team access the same storage area for shared variables.
reduction	The **reduction** clause performs a reduction on the scalar variables that appear in the list, with a specified operator.
default	The **default** clause allows the user to affect the data scope attributes of variables.

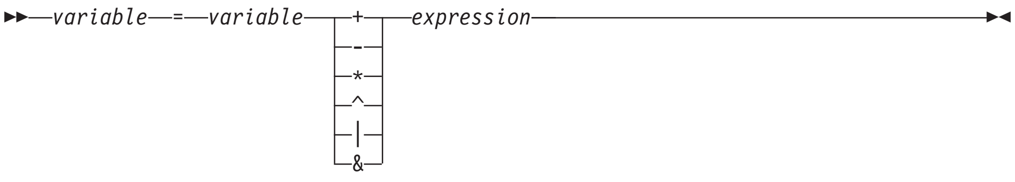
For more information, see the OpenMP directive descriptions in *Pragma directives for parallel processing* in the *XL C Compiler Reference* or the *OpenMP Application Program Interface Language Specification*.

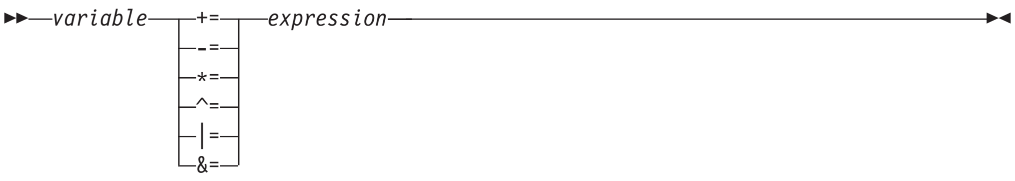
Related information in the *XL C Compiler Reference*

Pragma directives for parallel processing

Reduction operations in parallelized loops

The compiler can recognize and properly handle most reduction operations in a loop during both automatic and explicit parallelization. In particular, it can handle reduction statements that have either of the following forms:





where:

variable
: is an identifier designating an automatic or register variable that does not have its address taken and is not referenced anywhere else in the loop, including all loops that are nested. For example, in the following code, only S in the nested loop is recognized as a reduction:

```
int i,j, S=0;
for (i= 0 ;i < N; i++) {
    S = S+ i;
     for (j=0;j< M; j++) {
        S = S + j;
    }
}
```

expression
: is any valid expression.

Recognized reductions are listed by the **-qinfo=reduction** option. When using IBM directives, use critical sections to synchronize access to all reduction variables not recognized by the compiler. OpenMP directives provide you with mechanisms to specify reduction variables explictly.

Chapter 12. Memory debug library functions

This appendix contains reference information about the XL 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 105 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 information in the *XL C Compiler Reference*

-qheapdebug

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.

Related information

"Functions for debugging memory heaps" on page 37

_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);
```

Purpose

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 values

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

Examples

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

```
/*  _debug_calloc.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_calloc 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_calloc.c.
Heap state was valid at line 9 of _debug_calloc.c.
Memory error detected at line 14 of _debug_calloc.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);
```

Purpose

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 values

There is no return value.

Examples

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);
```

Purpose

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 values

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

Examples

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(100000))) {
      puts("Could not allocate memory block.");
      exit(EXIT_FAILURE);
   }
   ptr = (char*)realloc(ptr, 10);
   _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);
```

Purpose

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 values

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

Examples

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);
```

Purpose

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 values

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.

Examples

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);
```

Purpose

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 values

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.

Examples

This example creates a heap and allocates memory from it, then uses _debug_heapmin 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);
```

Purpose

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 values

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

Examples

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);
```

Purpose

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 values

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.

Examples

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.
```

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);
```

Purpose

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 values

Returns a pointer to *dest*.

Examples

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_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);
```

Purpose

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 values

Returns a pointer to *dest*.

Examples

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

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.

Purpose

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

Return values

Returns a pointer to the concatenated string *string1*.

Examples

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);
```

Purpose

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 values

Returns a pointer to the copied string *string1*.

Examples

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);
```

Purpose

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 values

Returns a pointer to the joined string *string1*.

Examples

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);
```

Purpose

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 values

Returns a pointer to *string1*.

Examples

This example contains a programming error. The source string is too long for the destination buffer, and the `strncpy` operation damages the heap.

```
/*  _debug_strncopy  */
#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\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);
```

Purpose

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 values

Returns a pointer to the altered *string*. There is no error return value.

Examples

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);
```

Purpose

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 values

Returns a pointer to the altered string. There is no error return value.

Examples

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

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

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1998, 2008. All rights reserved.

Trademarks and service marks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A complete and current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

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

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

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

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

Index

Program Number: 5724-U80

Printed in USA

SC23-8883-00