IBM XL C for AIX, V13.1.2

Optimization and Programming Guide

Version 13.1.2

SC27-4241-01

IBM XL C for AIX, V13.1.2

Optimization and Programming Guide

Version 13.1.2

SC27-4241-01

Note

Before using this information and the product it supports, read the information in "Notices" on page 139.

First edition

This edition applies to IBM XL C for AIX, V13.1.2 (Program 5765-J06; 5725-C71) 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 document

This guide discusses advanced topics related to the use of IBM® XL C for AIX®, V13.1.2, 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 document

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 document

This document uses a task-oriented approach to present 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, V13.1.2 documentation set, which provides detailed descriptions of compiler options and pragmas, and specific language extensions.

How this document is organized

This guide includes the following chapters:

- Chapter 1, "Using 32-bit and 64-bit modes," on page 1 discusses common problems that arise when you port existing 32-bit applications to 64-bit mode, and it provides recommendations for avoiding these problems.
- Chapter 2, "Using XL C with Fortran," on page 7 discusses considerations for calling Fortran code from XL C programs.
- Chapter 3, "Aligning data," on page 13 discusses options available for controlling the alignment of data in aggregates, such as structures.
- Chapter 4, "Handling floating-point operations," on page 21 discusses options available for controlling the way floating-point operations are handled by the compiler.
- Chapter 5, "Using memory heaps," on page 27 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 39 discusses how to compile and link static and shared libraries.
- Chapter 7, "Optimizing your applications," on page 43 discusses various options provided by the compiler for optimizing your programs, and it provides recommendations for use of these options.
- Chapter 8, "Debugging optimized code," on page 75 discusses the potential usability problems of optimized programs and the options that can be used to debug the optimized code.

- Chapter 9, “Coding your application to improve performance,” on page 81 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 95 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 111 provides an overview of options offered by XL C for creating multi-threaded programs, including IBM SMP and OpenMP language constructs.
- Chapter 12, “Memory debug library functions,” on page 121 provides a reference listing and examples of all compiler debug memory library functions.

Conventions

Typographical conventions

The following table shows the typographical conventions used in the IBM XL C for AIX, V13.1.2 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.
underlining	The default setting of a parameter of a compiler option or directive.	nomaf | maf
`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`.

Qualifying elements (icons)

In descriptions of language elements where a feature is exclusive to the C11 standard, or where a feature is an IBM extension of the C standard, this information uses icons to delineate segments of text as follows:

Table 2. Qualifying elements

Qualifier/Icon	Meaning
IBM extension begins ► IBM IBM ◄ IBM extension ends	The text describes a feature that is an IBM extension to the standard language specifications.
C11 begins ► C11 C11 ◄ C11 ends	The text describes a feature that is introduced into standard C as part of C11.

Syntax diagrams

Throughout this information, diagrams illustrate XL C syntax. This section helps 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:

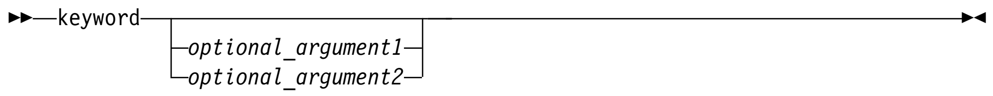
```
►►──keyword──┬────────────────────┬─────────────────────────────────►◄
             └─optional_argument──┘
```

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

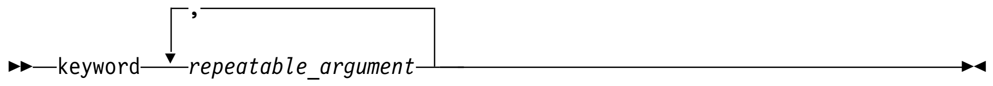
  If you *must* choose one of the items, one item of the stack is shown on the main path.

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

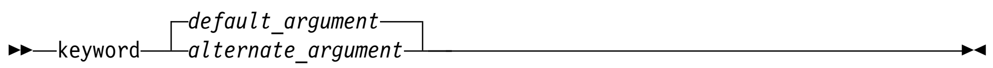
  If choosing one of the items is optional, the entire stack is shown below the main path.



- 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:



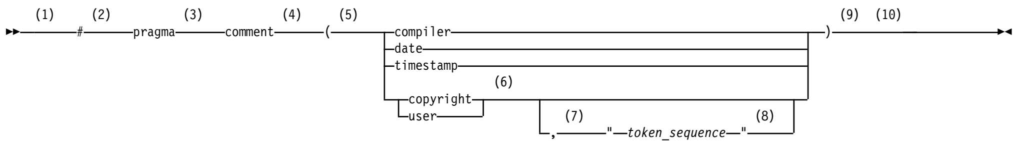
- The item that is the default is shown above the main path.



- 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")
```

Example of a syntax statement

EXAMPLE *char_constant* {*a*|*b*}[*c*|*d*]*e*[,*e*]... *name_list*{*name_list*}...

The following list explains the syntax statement:
- Enter the keyword EXAMPLE.
- Enter a value for *char_constant*.
- Enter a value for *a* or *b*, but not for both.
- Optionally, enter a value for *c* or *d*.
- Enter at least one value for *e*. If you enter more than one value, you must put a comma between each.
- Optionally, enter the value of at least one *name* for *name_list*. If you enter more than one value, you must put a comma between each *name*.

Note: The same example is used in both the syntax-statement and syntax-diagram representations.

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:
- Quick Start Guide

  The Quick Start Guide (`quickstart.pdf`) is intended to get you started with IBM XL C for AIX, V13.1.2. It is located by default in the XL C directory and in the `\quickstart` directory of the installation DVD.
- 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 DVD.
- 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, V13.1.2 Installation Guide*.
- Online product documentation

The fully searchable HTML-based documentation is viewable in IBM Knowledge Center at http://www.ibm.com/support/knowledgecenter/SSGH2K_13.1.2/com.ibm.compilers.aix.doc/welcome.html.

- PDF documents

  PDF documents are available on the web at .

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

Table 3. XL C PDF files

Document title	PDF file name	Description
IBM XL C for AIX, V13.1.2 Installation Guide, SC27-4238-01	`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, V13.1.2, SC27-4237-01	`getstart.pdf`	Contains an introduction to the XL C product, with information about setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
IBM XL C for AIX, V13.1.2 Compiler Reference, SC27-4239-01	`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, V13.1.2 Language Reference, SC27-4240-01	`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, V13.1.2 Optimization and Programming Guide, SC27-4241-01	`proguide.pdf`	Contains information about 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 Adobe Reader. If you do not have Adobe Reader, you can download it (subject to license terms) from the Adobe website at http://www.adobe.com.

More information related to XL C, including IBM Redbooks® publications, white papers, and other articles, is available on the web at http://www.ibm.com/support/docview.wss?uid=swg27036590.

For more information about C/C++, see the C/C++ café at https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=5894415f-be62-4bc0-81c5-3956e82276f3.

Standards and specifications

XL C is designed to support the following standards and specifications. You can refer to these standards and specifications 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 - C, ISO/IEC 9899:2011*, also known as *C11*. (Partial support)
- *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.
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985*.
- *OpenMP Application Program Interface Version 3.1* (full support) and *OpenMP Application Program Interface Version 4.0 (partial support)*, available at http://www.openmp.org

Other IBM information

- *Parallel Environment for AIX: Operation and Use*
- The IBM Systems Information Center, at http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/com.ibm.aix.doc/doc/base/aixparent.htm, is a resource for AIX information.

  You can find the following books for your specific AIX system:
  - *AIX Commands Reference, Volumes 1 - 6*
  - *Technical Reference: Base Operating System and Extensions, Volumes 1 & 2*
  - *AIX National Language Support Guide and Reference*
  - *AIX General Programming Concepts: Writing and Debugging Programs*
  - *AIX Assembler Language Reference*

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/support/entry/portal/overview/software/rational/xl_c_for_aix. 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 an email to compinfo@cn.ibm.com.

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

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 email to compinfo@cn.ibm.com.

Be sure to include the name of the manual, the part number of the manual, 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 the XL C compiler to develop either 32-bit or 64-bit applications. To do so, specify **-q32** or **-q64**, respectively, during compilation. Alternatively, you can set the *OBJECT_MODE* environment variable to 32 or 64 at compile time. If both *OBJECT_MODE* and **-q32**/**-q64** are specified, **-q32**/**-q64** takes precedence.

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 4. 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, signed 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 (defined in the header file <cstddef>)	4 bytes	4-byte boundaries	8 bytes	8-byte boundaries
ptrdiff_t (defined in the header file <cstddef>)	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, will occur when the program is executed.

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 that are 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 5. 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 (largest 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 (largest 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

These differences have the following implications:

- Assigning a `long` value to a `double` variable can cause loss of accuracy.
- Assigning constant values to `long` 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. These operations can impact performance.

In situations where a `long` value can overflow when assigned to other variables or passed to functions, you must observe the following guidelines:

- Avoid implicit type conversion by using explicit type casting to change types.
- Ensure that all functions that accept or return `long` types are properly prototyped.
- Ensure that `long` type 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 twos complement representation to truncate values that 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 the following boundary areas:

- constant > `UINT_MAX`
- constant < `INT_MIN`
- constant > `INT_MAX`

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

Table 6. 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` returns 8 bytes. Similar problems occur when the compiler passes constants directly to functions.

You can avoid these problems by using the suffixes `L` (for long constants), `UL` (for unsigned long constants), `LL` (for long long constants), or `ULL` (for unsigned long 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 in the preceding paragraph, suffixing the number as `4294967295U` forces the compiler to always recognize the constant as an `unsigned int` in 32-bit or 64-bit mode. These suffixes can also be applied to hexadecimal constants.

Bit-shifting long values

Left-bit-shifting long values produces different results in 32-bit and 64-bit modes. The examples in Table 7 show the effects of performing a bit-shift on long constants using the following code segment:

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

Table 7. Results of bit-shifting long values

Initial value	Symbolic constant	Value after bit shift by one bit	
		32-bit mode	64-bit mode
0x7FFFFFFFL	INT_MAX	0xFFFFFFFE	0x00000000FFFFFFFE
0x80000000L	INT_MIN	0x00000000	0x0000000100000000
0xFFFFFFFFL	UINT_MAX	0xFFFFFFFE	0x00000001FFFFFFFE

In 32-bit mode, `0xFFFFFFFE` is negative. In 64-bit mode, `0x00000000FFFFFFFE` and `0x00000001FFFFFFFE` are both positive.

Assigning pointers

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

- 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.

In C, the following code is valid in 32-bit mode without a prototype:

```
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 does not prevent the truncation, as the address of the memory allocated by `calloc` was already truncated during the return. In this example, the best solution is to include the header file, `stdlib.h`, which contains the prototype for `calloc`. An alternative solution is to prototype the function as it is in the header file.

To avoid these types of problems, you can take the following measures:

- Prototype any functions that return a pointer, where possible by using the appropriate header file.
- Ensure 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.
- Use the **`-qwarn64`** option to get warning messages in the listing file about potential problems.

Aligning aggregate data

Normally, 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 need to 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.

Any aggregate data written to a file in one mode cannot be correctly read in the other mode. Data exchanged with other languages has similar problems.

For detailed information about aligning data structures, including structures that contain bit fields, see Chapter 3, “Aligning data,” on page 13.

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 on the Fortran side of such applications. The following table lists C types and the equivalent Fortran types in the different modes.

Table 8. Equivalent C and Fortran data types

C type	Fortran type	
	32-bit	**64-bit**
signed int	INTEGER	INTEGER

Table 8. Equivalent C and Fortran data types (continued)

C type	**Fortran type**	
	32-bit	**64-bit**
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 7

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 10
- "Function calls and parameter passing" on page 10
- "Pointers to functions" on page 11

In topic "Sample program: C calling Fortran" on page 11, an example of a C program that calls a Fortran subroutine is provided.

For more information about language interoperability, see the information about the BIND attribute and the interoperability of procedures in the *XL Fortran Language Reference*.

Related information:

"Calling Fortran code" on page 4

Identifiers

You need to follow these recommendations when writing C code to call functions that are 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].

Note:

1. The Fortran 90 and 95 language standards require identifiers to be no more than 31 characters; the Fortran 2003 and the Fortran 2008 standards require identifiers to be no more than 63 characters.

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 you program for interlanguage calls.

Table 9. Correspondence of data types between C and Fortran

C data types	**Fortran data types**	
	Types	**Types with kind type parameters from the ISO_C_BINDING module**
_Bool	LOGICAL*1 or LOGICAL(1)	LOGICAL(C_BOOL)
char	CHARACTER	CHARACTER(C_CHAR)

Table 9. Correspondence of data types between C and Fortran (continued)

C data types	**Fortran data types**	
	Types	**Types with kind type parameters from the ISO_C_BINDING module**
signed char	INTEGER*1 or INTEGER(1)	INTEGER(C_SIGNED_CHAR)
unsigned char	LOGICAL*1 or LOGICAL(1)	
signed short int	INTEGER*2 or INTEGER(2)	INTEGER(C_SHORT)
unsigned short int	LOGICAL*2 or LOGICAL(2)	
int	INTEGER*4 or INTEGER(4)	INTEGER(C_INT)
unsigned int	LOGICAL*4 or LOGICAL(4)	
signed long int	INTEGER*4 or INTEGER(4) (with **-q32**), INTEGER*8 or INTEGER(8) (with **-q64**)	INTEGER(C_LONG)
unsigned long int	LOGICAL*4 or LOGICAL(4) (with **-q32**), INTEGER*8 or INTEGER(8) (with **-q64**)	
signed long long int	INTEGER*8 or INTEGER(8)	INTEGER(C_LONG_LONG)
unsigned long long int	LOGICAL*8 or LOGICAL(8)	
size_t	INTEGER*4 or INTEGER(4) (with **-q32**), INTEGER*8 or INTEGER(8) (with **-q64**)	INTEGER(C_SIZE_T)
intptr_t	INTEGER*4 or INTEGER(4) (with **-q32**), INTEGER*8 or INTEGER(8) (with **-q64**)	INTEGER(C_INTPTR_T)
intmax_t	INTEGER*8 or INTEGER(8)	INTEGER(C_INTMAX_T)
int8_t	INTEGER*1 or INTEGER(1)	INTEGER(C_INT8_T)
int16_t	INTEGER*2 or INTEGER(2)	INTEGER(C_INT16_T)
int32_t	INTEGER*4 or INTEGER(4)	INTEGER(C_INT32_T)
int64_t	INTEGER*8 or INTEGER(8)	INTEGER(C_INT64_T)
int_least8_t	INTEGER*1 or INTEGER(1)	INTEGER(C_INT_LEAST8_T)
int_least16_t	INTEGER*2 or INTEGER(2)	INTEGER(C_INT_LEAST16_T)

Table 9. Correspondence of data types between C and Fortran (continued)

C data types	**Fortran data types**	
	Types	**Types with kind type parameters from the ISO_C_BINDING module**
int_least32_t	INTEGER*4 or INTEGER(4)	INTEGER(C_INT_LEAST32_T)
int_least64_t	INTEGER*8 or INTEGER(8)	INTEGER(C_INT_LEAST64_T)
int_fast8_t	INTEGER, INTEGER*4, or INTEGER(4)	INTEGER(C_INT_FAST8_T)
int_fast16_t	INTEGER*4 or INTEGER(4)	INTEGER(C_INT_FAST16_T)
int_fast32_t	INTEGER*4 or INTEGER(4)	INTEGER(C_INT_FAST32_T)
int_fast64_t	INTEGER*8 or INTEGER(8)	INTEGER(C_INT_FAST64_T)
float	REAL, REAL*4, or REAL(4)	REAL(C_FLOAT)
double	REAL*8, REAL(8), or DOUBLE PRECISION	REAL(C_DOUBLE)
long double (default)	REAL*8, REAL(8), or DOUBLE PRECISION	REAL(C_LONG_DOUBLE)
long double (with -qlongdouble or -qldbl128)	REAL*16 or REAL(16)	REAL(C_LONG_DOUBLE)
float _Complex	COMPLEX*4, COMPLEX(4), COMPLEX*8, or COMPLEX(8)	COMPLEX(C_FLOAT_COMPLEX)
double _Complex	COMPLEX*8, COMPLEX(8), COMPLEX*16, or COMPLEX(16)	COMPLEX(C_DOUBLE_COMPLEX)
long double _Complex (default)	COMPLEX*16, COMPLEX(16), COMPLEX*32, or COMPLEX(32)	COMPLEX(C_LONG_DOUBLE_COMPLEX)
long double _Complex (with -qlongdouble or -qldbl128)	COMPLEX*16, COMPLEX(16), COMPLEX*32, or COMPLEX(32)	COMPLEX(C_LONG_DOUBLE_COMPLEX)
struct or union	derived type	
enum	INTEGER*4 or INTEGER(4)	
char[n]	CHARACTER*n or CHARACTER(n)	
array pointer to type, or type []	Dimensioned variable (transposed)	
pointer to function	Functional parameter	

Table 9. Correspondence of data types between C and Fortran (continued)

C data types	Fortran data types	
	Types	Types with kind type parameters from the ISO_C_BINDING module
struct with -qalign=packed	Sequence derived type	

Related information in the *XL C Compiler Reference*

-qldbl128, -qlongdouble

-qalign

Character and aggregate data

Most numeric data types have counterparts across C and 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.
- An n-element C array is indexed with `0...n-1`, whereas an n-element Fortran array is typically indexed with `1...n`. In addition, Fortran supports user-specified bounds while C does not.
- 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). The following table shows how a two-dimensional array declared by `A[3][2]` in C and by `A(3,2)` in Fortran, is stored:

Table 10. 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 equivalently in both C and 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.

For more information about interlanguage calls to functions or routines, see the information about interlanguage calls in the *XL Fortran Optimization and Programming Guide*.

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. Starting from the Fortran 2003 standard, Fortran variables of type C_FUNPTR are interoperable with function pointers. 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. The example includes the following source files:

- The main program source file: `example.c`
- The Fortran add function source file: `add.f`

Main program source file: `example.c`

```
#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);
}
```

Fortran add function source file: `add.f`

```
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
```

Compile the main program and Fortran add function source files as follows:

```
xlc -c example.c
xlf -c add.f
```

Link the object files from compile step to create executable `add`:

```
xlc -o add example.o add.o
```

Execute binary:

```
./add
```

The output is as follows:

```
The sum of 3 and 7 is 10
```

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 need to take into account the differences between alignment settings available in 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. For more information, see the *AltiVec Technology Programming Interface Manual*.

XL C provides alignment modes and alignment modifiers for specifying data alignment. Using alignment modes, you can set alignment defaults for all data types for a compilation unit (or subsection of a compilation unit) by specifying a predefined suboption.

Using alignment modifiers, you can 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 different platforms and addressing models, the suboptions and pragmas that 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 different alignment modes.

"Using alignment modifiers" on page 19 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 that govern 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 that is 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 as follows:

- 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 11 on page 14.
- 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 11. 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 11, which provides the alignment value, in bytes, for scalar variables of 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 11. 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

Notes:

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.

Notes:

- Vectors in a bit-packed structure might not be correctly aligned unless you take extra action to ensure their alignment.
- Vectors might suffer from alignment issues if they are accessed through heap-allocated storage or through pointer arithmetic. For example, `double my_array[1000] __attribute__((__aligned__(16)))` is 16-byte aligned while `my_array[1]` is not. How `my_array[i]` is aligned is determined by the value of `i`.

"Alignment of aggregates" discusses the rules for the alignment of entire aggregates and provides examples of aggregate layouts. "Alignment of bit-fields" on page 17 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, -qlongdouble

#pragma options

Alignment of aggregates

The data contained in Table 11 on page 14 (in "Using alignment modes" on page 13) apply to scalar variables, and variables that are members of aggregates such as structures, unions, and classes. 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. As a result, two distinct variables might have the same address.
- 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 12. Alignment and aggregate size

Example	Size of aggregate		
	-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.)
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.)

Notes:

- The alignment of an aggregate must be the same in all compilation units. For example, if the declaration of an aggregate is in a header file and you include that header file into two distinct compilations units, choose the same alignment mode for both compilations units.

For rules on the alignment of aggregates containing bit fields, see "Alignment of bit-fields" on page 17.

Alignment examples

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

`p` = padding

`|` = halfword (2-byte) boundary

`:` = byte boundary

Mac68K example

```
#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 as follows:

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

Packed example

```
#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 as follows:

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

Nested aggregate example

```
#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 as follows:

```
|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`, `int`, `unsigned int`, `long`, or `unsigned long` data type. The alignment of a bit-field depends on its base type and the compilation mode (32-bit or 64-bit).

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 returns the size of the base type.

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, which moves 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

```
#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, with which you can exercise even finer-grained control over alignment, at the level of declaration or definition of individual variables or aggregate members. Available modifiers are as follows:

#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**.
- 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. 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 might effectively increase 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.

Notes:

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 the information about type attributes in the *XL C Language Reference*.

Related information in the *XL C Compiler Reference*

#pragma pack

Related information in the *XL C Language Reference*

The aligned type attribute (IBM extension)

The packed type attribute (IBM extension)

The __align type qualifier (IBM extension)

Type attributes (IBM extension)

The aligned variable attribute (IBM extension)

The packed variable attribute (IBM extension)

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" on page 22
- "Compiling for strict IEEE conformance" on page 22
- "Handling floating-point constant folding and rounding" on page 23
- "Handling floating-point exceptions" on page 25

Floating-point formats

XL C supports the following binary floating-point formats:

- 32-bit single precision, with an 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 normalized range of 0 and 10^{-308} to 10^{308} and precision of about 16 decimal digits
- 128-bit extended precision, with slightly greater range than double-precision values, and with a precision of about 32 decimal digits

Note that the `long double` type might represent either double-precision or extended-precision values, depending on the setting of the **`-qldbl128`** compiler option.

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

XL C extended precision is not in the **binary128** format that is suggested by the IEEE standard. The IEEE standard suggests that extended formats use more bits in the exponent for greater range and the fraction for greater precision.

Special numbers, such as NaN, infinity, and negative zero, are not fully supported by the 128-bit extended precision values. Arithmetic operations do not necessarily propagate these numbers in extended precision.

Related information in the *XL C Compiler Reference*

-qldbl128, -qlongdouble

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 might also produce a more precise result. However, the increased precision might lead to different results from those obtained in other environments, and might 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, use the following options:

- Use the **-qfloat=nomaf** compiler option.
- If the program changes the rounding mode at run time, use the **-qfloat=rrm** option.
- If the data or program code contains signaling NaN values (NaNS), use any of the following groups of options. (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.)
  - The **-qfloat=nans** and **-qstrict=nans** options
  - The **-qfloat=nans** and **-qstrict** options
- If you compile with **-O3**, **-O4**, or **-O5**, include the option **-qstrict** after it. You can also use the suboptions of **-qstrict** to refine the level of control for the transformations performed by the optimizers.
- If you use AIX operating system functions to enable hardware trapping on floating-point exceptions, use the **-qfloat=fenv** option to tell the optimizer that traps can occur.

Related information:

"Advanced optimization" on page 46

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 might 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, perform either of the following operations:
  - 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."
  - 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, perform either of the following operations:
  - 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, NaN, or underflow to zero 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 25

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 run time is round-to-nearest, ties to even. If your program changes the rounding mode at run time, 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;
}
```

Notes:

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, -infinity, , zero , or 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 run time. However, you can use the **-qflttrap** option or call C or operating system functions to detect these types of exceptions. If you enable floating-point traps without using the **-qflttrap** option, use the **-qfloat=fenv** option.

In addition, you can add suitable support code to your program to make program execution continue after an exception occurs and to modify the results of operations causing exceptions.

Because, however, floating-point computations involving constants are usually folded at compile time, the potential exceptions that would be produced at run time might not occur. To ensure that the **-qflttrap** option traps all runtime floating-point exceptions, you can use the **-qfloat=nofold** option to suppress all compile-time folding.

If you use the AIX operating system functions to enable hardware trapping on floating-point exceptions, use the **-qfloat=fenv** option to inform the compiler that exceptions might occur.

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 and define the __STDC_WANT_DEC_FP__ macro when you compile them.

For example, to compile `dfp_hello.c`, use the following compiler invocation:

```
xlc dfp_hello.c -qdfp -qarch=pwr7 -D__STDC_WANT_DEC_FP__
```

```
#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);
 }
```

Besides defining the __STDC_WANT_DEC_FP__ macro on the command line, you can also define this macro in your source files using the `#define` directive.

Related information in the *XL C Compiler Reference*

-qdfp

-D

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. With these functions, you can perform the following tasks:

- 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 34 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 (the memory and swapper size of your system 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 can 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 with which you can 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 13. 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 29.
n/a	_uopen	Opens a heap for use by a process. Described in “Using a heap” on page 31.
n/a	_ustats	Provides information about a heap. Described in “Getting information about a heap” on page 32.
n/a	_uaddmem	Adds memory blocks to a heap. Described in “Expanding a heap” on page 30.
n/a	_uclose	Closes a heap from further use by a process. Described in “Closing and destroying a heap” on page 32.
n/a	_udestroy	Destroys a heap. Described in “Closing and destroying a heap” on page 32.
calloc	_ucalloc	Allocates and initializes memory from a heap you have created. Described in “Using a heap” on page 31.
malloc	_umalloc	Allocates memory from a heap you have created. Described in “Using a heap” on page 31.
_heapmin	_uheapmin	Returns unused memory to the system. Described in “Closing and destroying a heap” on page 32.

Table 13. 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 33.

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. When 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 do not 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, call the _uaddmem function.
- For dynamically-sized heaps only, write 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.
- *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

After 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 do not 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 gives you the following information:

- 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`. After you have closed the heap in a process, that process can no longer allocate memory 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 the following section.)

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
```

Example of a user heap with regular memory

The following program 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;
}
```

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 35.)
- Debug versions of all memory management functions. (Described in "Functions for debugging memory heaps" on page 35.)

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 whether the heap is consistent; they do not provide the details as 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 do not 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 121

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 any 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 14. 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 33.

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 Chapter 12, "Memory debug library functions," on page 121.

Table 15. Functions for debugging memory heaps

Default heap function	**Corresponding user-created heap function**
_debug_calloc	_debug_ucalloc

Table 15. Functions for debugging memory heaps (continued)

Default heap function	Corresponding user-created heap function
_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 want 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 33.

Notes:

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

Additional functions for debugging memory heaps

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

Table 16. 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`.
_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 121

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.

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:

Objects and libraries

Compiling and linking a library

This section describes how to compile your source files into object files for inclusion in a library, how to link a library into the main program, and how to link one library into another.

Related information:

Diagnosing link-time errors

Dynamic and static linking

Compiling a static library

To compile a static (unshared) library, follow this procedure:

1. Compile each source file to get an object file. For example:
   ```
   xlc -c test.c example.c
   ```
2. Use the ar command to add the generated object files to an archive library file. For example:
   ```
   ar -rv libex.a test.o example.o
   ```

Compiling a shared library

For use with dynamic linking

To compile a shared library that uses dynamic linking:

1. Compile each source file into an object file, with no linking. For example:
   ```
   xlc -c test1.c -o test1.o
   ```
2. Optional: 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 41.
   - Use the **-qexpfile** compiler option with the **-qmkshrobj** option. The **-qexpfile** option saves all exported symbols from a list of given object files in a designated file. For example:
     ```
     xlc -qmkshrobj -qexpfile=exportlist test1.o test2.o
     ```
   - Manually create an export file using a text editor. You can edit an export file to include or exclude global symbols from the target shared library.
3. Use the **-qmkshrobj** option to create a shared library from the generated object files.
   - If you created an export file in step 2, use the **-bE** linker option to use your global symbol export list. If you do not specify a **-bE** option, all the global symbols are exported IBM except for those symbols that have the hidden or internal visibility attribute. IBM

   For example:

```
xlc -qmkshrobj -o mySharedObject.o test1.o test2.o -bE:exportlist
```

Notes:
- The default name of the shared object is **shr.o**, unless you use the **-o** option to specify another name.
- Exporting some functions (such as `restf#` where # is a number) might cause incorrect execution. If any files in the shared library use floating point and are compiled with the **-qtwolink** option, do not export the `restf#` or equivalent floating-point utility functions.

4. Optional: Use the AIX **ar** command to produce an archive library file from multiple shared or static objects. For example:
   ```
   ar -rv libtest.a mySharedObject.o myStaticObject.o
   ```
5. Link the shared library to the main application, as described in "Linking a library to an application" on page 42.

For use with runtime linking

To create a shared library that uses runtime linking:
1. Compile each source file into an object file with no linking.
2. Optional: 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 41.
   - Use the **-qexpfile** compiler option with the **-qmkshrobj** option. The **-qexpfile** option saves all exported symbols from a list of given object files in a designated file.
   - Manually create an export file using a text editor. You can edit an export file to include or exclude global symbols from the target shared library.
3. Use the **-G** option to create a shared library from the generated object files, and to enable runtime linking with applications that support it.
   - If you created an export file, use the **-bE** linker option to use your global symbol export list. If you do not specify a **-bE** option, all the global symbols are exported IBM except for those symbols that have the hidden or internal visibility attribute. IBM

   For example:
   ```
   xlc -G -o libtest.so test1.o test2.o -bE:exportlist
   ```
4. Link the shared library to the main application using the **-brtl** option, as described in "Linking a library to an application" on page 42.

Dynamic loading of a shared library

Shared libraries built for either dynamic or runtime linking can be dynamically loaded. See the AIX documentation for more information about using the dynamic loading routines:
- dlopen
- dlclose
- dlerror
- loadAndInit
- loadbind
- loadquery

- terminateAndUnload

Related external information

- *AIX Linking and Loading Mechanisms*
- **ar** and **ld** in the *AIX Commands Reference, Volumes 1 - 6*
- Shared Objects and Runtime Linking

Related information in the *XL C Compiler Reference*

-qexpfile

-qmkshrobj

-O, -qoptimize

-G

-qvisibility (IBM extension)

#pragma GCC visibility push, #pragma GCC visibility pop (IBM extension)

-brtl

Exporting symbols with the CreateExportList utility

CreateExportList is a shell script that creates a file containing a list of exportable 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 is 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.
- IBM All global symbols in the object files have the hidden or internal visibility attribute. IBM

Related information in the *XL C Compiler Reference*

-qmkshrobj

-qexpfile

-qvisibility (IBM extension)

#pragma GCC visibility push, #pragma GCC visibility pop (IBM extension)

Linking a library to an application

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

```
xlc -o myprogram main.c -Ldirectory1:directory2  -ltest
```

You instruct the linker to search for `libtest.a` in the first directory specified by the **-L** option. If `libtest.a` is not found, the search continues with the next directory specified by the **-L** option.

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

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

By using the **-brtl** option with the **-l** option, you instruct the linker to search for `libtest.so` in the first directory specified by the **-L** option. If `libtest.so` is not found, the linker searches for `libtest.a`. If neither file is found, the search continues with the next directory specified by the **-L** option.

For additional linkage options, including options that modify the default behavior, see the AIX **ld** documentation (http://publib.boulder.ibm.com/infocenter/aix/v7r1/index.jsp?topic=/com.ibm.aix.cmds/doc/aixcmds3/ld.htm).

Related information in the *XL C Compiler Reference*

-l

-L

-brtl

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 -ltest
```

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 run time, 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 bring the following benefits:

- Reduce the number of instructions that 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.

Each basic optimization technique can result in a performance benefit, although not all optimizations can benefit all applications. Consult the "Steps in the optimization process" on page 44 for an overview of the common sequence of steps that you can use to increase the performance of your application.

Tuning

Tuning is a developer-driven process where you experiment with changes, for example, to source code or compiler options, to allow the compiler to do a better job of optimizing your program. While optimization applies general transformations designed to improve the performance of any application in any supported environment, tuning offers you opportunities to adjust specific characteristics or target execution environments of your application to improve its performance. 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 make the following improvements:

- Select more efficient machine instructions.
- Generate instruction sequences that are more relevant to your application.
- Select from more focussed optimizations to improve your code.

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, 4, and 5.
3. The Debugging optimized 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 Table 17. The table also includes a column for compiler options that can have a performance benefit at that optimization level for some applications.

Table 17. Basic optimizations

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

Optimizing at level 0

Benefits at level 0

- Provides minimal performance improvement with minimal impact on machine resources
- Exposes some source code problems to help in the debugging process

Begin your optimization process at **-O0**, which the compiler already specifies by default. This level performs basic analytical optimization by removing obviously redundant code, and it can result in better compile time. It also ensures your code is algorithmically correct so you can move forward to more complex optimizations. **-O0** also includes some redundant instruction elimination and constant folding. The **-qfloat=nofold** option can be used to suppress folding floating-point operations.

Optimizing at this level accurately preserves all debugging 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 that your application takes advantage of all applicable architectural benefits.

Note: For SMP programs, you need to add an additional option **-qsmp=noopt**.

Related information in the *XL C Compiler Reference*

-qarch

Optimizing at level 2

Benefits at level 2

- Eliminates redundant code
- Performs basic loop optimization
- Structures code to take advantage of **-qarch** and **-qtune** settings

After you successfully compile, execute, and debug 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. By targeting the proper hardware, the optimizer can 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 that is consistent with the architecture choice and that can execute optimally on the chosen tuning hardware target. With this option, you can compile for a general set of targets and have the code run best on a particular target.

The **-O2** option can perform a number of additional optimizations as follows:

- 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.
- Global 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.
- Moving loop-invariant code out of loops.
- Simplifying control flow.

- Strength reduction and effective use of addressing modes.
- Widening, which merges adjacent load/stores and other operations.
- Pointer aliasing improvements to enhance other optimizations.

Even with **-O2** optimizations, some useful information about your source code is made available to the debugger if you specify **-g**. Using a higher **-g** level increases the information provided to the debugger but reduces the optimization that can be done. Conversely, higher optimization levels can transform code to an extent to which debugging information is no longer accurate.

Advanced optimization

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

After applying "Basic optimization" on page 44 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 that your application undergoes are largely under your control. The discussion of each optimization level in Table 18 includes information on the performance benefits and the possible trade-offs and information on how you can help guide the optimizer to find the best solutions for your application.

Table 18. 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**

When you compile programs with any of the following sets of options:
- **-O3 -qhot**
- **-O4**
- **-O5**

the compiler automatically attempts to vectorize calls to system math functions by calling the equivalent vector functions in the Mathematical Acceleration Subsystem libraries (MASS), with the exceptions of functions `vdnint`, `vdint`, `vcosisin`, `vscosisin`, `vqdrt`, `vsqdrt`, `vrqdrt`, `vsrqdrt`, `vpopcnt4`, and `vpopcnt8`. If the compiler cannot vectorize, it automatically tries to call the equivalent MASS scalar functions. For automatic vectorization or scalarization, the compiler uses versions of the MASS functions contained in the system library `libxlopt.a`.

In addition to any of the preceding sets of options, when the **-qipa** option is in effect, if the compiler cannot vectorize, it tries to inline the MASS scalar functions before deciding to call them.

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

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 setting the default 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, because **-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
- Using alternative calculations that might give slightly less precise results or not handle infinities or NaNs in the same way

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” on page 44. 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.

For information on the **-O** level syntax, see "-O -qoptimize" in the *XL C Compiler Reference* .

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**.

Conversely, if the application does not use loops processing arrays (which **-qhot** improves), you can improve compile speed significantly, usually with minimal performance loss by using **-qnohot** after **-O3**.

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 time and link time.

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 the execution environment of your application, you must specify a different **-qarch** suboption after the **-O4** option. This overrides **-qtune=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 at a time.

  If you want to execute your application on a different machine, specify correct cache values.

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 compile 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.

Optimizing at level 5

Benefits at level 5

- Makes 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 the 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 levels, 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 19. 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 50 for more information about this option.

Table 19. Target machine options (continued)

Option	Behavior
-qtune	Biases optimization toward execution on a given microprocessor, without implying anything about the instruction set architecture to be used as a target. See "Getting the most out of target machine options" for more information about 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" for more information about this option.

For a complete list of valid hardware related suboptions and combinations of suboptions, see the following information in the *XL C Compiler Reference*.

- *Acceptable -qarch/-qtune combinations* in the -qtune section.
- The Specifying compiler options for architecture-specific compilation section.

Related information in the *XL C Compiler Reference*

📄 -q32, -q64

📄 -qarch

📄 -qipa

📄 -qcache

Getting the most out of target machine options

Using -qarch options

Use the **-qarch** compiler option to generate instructions that are optimized for a specific machine architecture. For example, if you want to generate an object code that contains instructions optimized for POWER8®, you use **-qarch=pwr8**. If your application runs 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, use the **-qarch** option to specify the smallest possible family of the machines that can run your code reasonably well.

If you want to run your application on a system architecture that provides specific feature supports, you must specify a corresponding **-qarch** suboption to generate the object code for your system architecture. For example, if you want to deploy your application on a POWER7®, POWER7+™, or POWER8 machine and to fully exploit VMX vector processing and large-page support, you must specify **-qarch=pwr7** for POWER7 or POWER7+ and **-qarch=pwr8** for POWER8 on your compiling machine. Specifying **-qarch=auto** or **-qarch** does not give you the support you want. However, if you deploy your application on both POWER7 and POWER8, you must make sure that **-qarch** is set to the lowest common architecture. This way your application will only contain instructions that are common to all processors the application is deployed on. In this example, the lowest common architecture is POWER7, so you must use **-qarch=pwr7**. For details about **-qarch** and its suboptions, see -qarch in the *XL C Compiler Reference*. For details about the corresponding system architectures each **-qarch** suboption supports, see the Features support in processor architectures table in -qarch.

Using -qtune options

Use the **-qtune** compiler option to control the scheduling of instructions that are optimized for your machine architecture. If you specify a particular architecture with **-qarch**, **-qtune** automatically selects 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** generates code that runs on all of the architectures in the specified group, but the instruction sequences are 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 about the valid combinations of **-qarch** and **-qtune** settings, see *Acceptable -qarch/-qtune combinations* in the **-qtune** section of the *XL C Compiler Reference*.

If you need to create a single binary file that runs on a range of PowerPC hardware, you can use 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 might be harmful on some hardware.

Note: You must verify the performance of code compiled with the **-qtune=balanced** option before distributing it.

The difference between **-qtune=balanced** and other **-qtune** suboptions including **-qtune=auto** is as follows:
- With the **-qtune=balanced** option, the compiler generates instructions that perform reasonably well across a range of Power® hardware.
- With other suboptions, the compiler generates instructions that are optimized for that specified versions of hardware architecture and might not perform well on others.

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.

Related information in the *XL C Compiler Reference*

-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 address 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 SIMD vector instructions
- Generating calls to vector math library functions

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

Table 20. -qhot suboptions

Suboption	Behavior
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** and **-qhot=noarraypad**. This level is automatically enabled if you compile with **-O3**.
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**.
level=2	When used with **-qsmp**, instructs the compiler to perform the transformations of **-qhot=level=1** plus some additional transformation on nested loops. The resulting loop analysis and transformations can lead to more cache reuse and loop parallelization.
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.

Related information in the *XL C Compiler Reference*

-qhot

-qstrict

-qignerrno

-qarch

-qsimd

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. However, it increases compilation time and might have little benefit if the program has no loop processing vectors or arrays. In this case, using **-O3 -qnohot** might be better.
- 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 compilation time (this can happen with complex loop nests), try **-qhot=level=0** or **-qnohot**.
- If your code size is unacceptably large, try reducing the inlining level or using **-qcompact** along with **-qhot**.
- You can compile some source files with the **-qhot** option and some files without the **-qhot** option, allowing the compiler to improve only the parts of your code that need optimization.
- Use **-qreport** along with **-qsimd=auto** to generate a loop transformation listing. The listing file identifies how loops are 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 might 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 might prevent loop vectorization.
- Use **-qreport** along with **-qhot** or any optimization option that implies **-qhot** to generate information about nested loops in the `LOOP TRANSFORMATION SECTION` of the listing file. In addition, when you use **-qprefetch=assistthread** to generate prefetching assist threads, a message `Assist thread for data prefetching was generated` is also displayed in this section of the report. To generate a list of aggressive loop transformations and parallelizations performed on loop nests in the `LOOP TRANSFORMATION SECTION` of the listing file, use **-qhot=level=2** and **-qsmp** together with **-qreport**.
- Use **-qassert=refalign**, where appropriate, to assert to the compiler that all pointers inside the compilation unit only point to data that is naturally aligned with respect to the length of the pointer types. With this assertion, the compiler might generate more efficient code. This assertion is particularly useful when you target a SIMD architecture with **-qhot=level=0** or **-qhot=level=1** with the **-qsimd=auto** option.

Related information in the *XL C Compiler Reference*

-qcompact

-qhot

-qsimd

-qprefetch

-qstrict

Using shared-memory parallelism (SMP)

Most 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 **-qsmp** option implies the **-qhot** option and an optimization level of **-O2** or higher.

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 111.

Table 21. 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 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, locking, and so on.

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 thread-safe 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 it can result in significant performance improvements.

You can specify interprocedural analysis on the compilation step only or on both compilation and link steps in whole program mode. Whole program mode expands the scope of optimization to an entire program unit, which can be an executable or a shared object. As IPA can significantly increase compilation time, you should limit using IPA to the final performance tuning stage of development.

You can generate relinkable objects while preserving IPA information by specifying **-r -qipa=relink**. This creates a nonexecutable package that contains all object files. By using this suboption, you can postpone linking until the very last stage.

If you want to use your own archive files while generating the nonexecutable package, you can use the **ar** tool and set the *XL_AR* environment variable to point to the **ar** tool. For details, refer to the -qipa section of the *XL C Compiler Reference*.

You can 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 as follows:

1. Do preliminary performance analysis and tuning before compiling with the **-qipa** option, because the IPA analysis uses a two-pass mechanism that increases compilation time and link time. You can reduce some compilation and link overhead by using the **-qipa=noobject** option.
2. Specify the **-qipa** option on both the compilation 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 22. 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.
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*	Provides precise control over function inlining.
fine_tuning	Other values for **-qipa** provide the ability to specify the behavior of library code, tune program partitioning, read commands from a file, and so on.
relink	Creates a nonexecutable package that contains all of your object files while preserving IPA information.

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 the 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, use the compiler driver (**xlc**) to link with all the compiler options on the link step included.

- 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 previously inlined. If too few or too many functions are inlined, consider using **`-qinline`** or **`-qnoinline`**. To control the inlining of specific functions, use **-qinline**+*function_name* or **-qinline**-*function_name*.
- To generate data reorganization information in the listing file, specify the optimization level **`-qipa=level=2`** or **`-O5`** together with **`-qreport`**. During the IPA link pass, the data reorganization messages for program variable data will be produced to the data reorganization section of the listing file with the label `DATA REORGANIZATION SECTION`. Reorganizations include array splitting, array transposing, memory allocation merging, array interleaving, and array coalescing.
- Use **`-r -qipa=relink`** to create a nonexecutable package that contains all of your object files while preserving IPA information. If you want to use your archive files while generating the package, you can use the **`ar`** tool and set the *XL_AR* environment variable to point to the **`ar`** tool. For details, refer to the section of the *XL C Compiler Reference*.

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 that might have previously modified unused memory might now conflict with user-allocated storage.
- Dereferencing a pointer that has been cast to an incompatible type.

Related information in the *XL C Compiler Reference*

-qinline

-qlist

-qipa

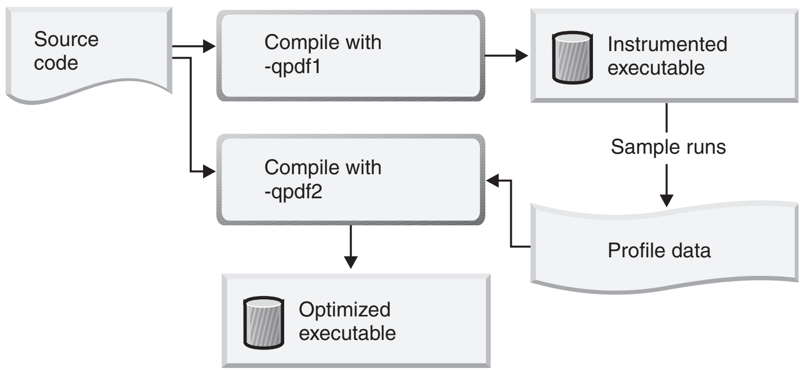
Using profile-directed feedback

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 run.

Use the PDF process as one of the last steps of optimization before putting the application into production. Optimization at all levels from **-O2** up can benefit from PDF. Other optimizations such as the **-qipa** option and optimization levels **-O4** and **-O5** can also benefit from PDF process.

The following diagram illustrates the PDF process.

Figure 1. Profile-directed feedback



To use the PDF process to optimize your application, follow these steps:

1. Compile some or all of the source files in a program with the **-qpdf1** option. You must specify at least the **-O2** optimization level.

   Notes:

   - A PDF map file is generated at this step. It is used for the **showpdf** utility to display part of the profiling information in text or XML format. For details, see "Viewing profiling information with showpdf" on page 61. If you do not need to view the profiling information, specify the **-qnoshowpdf** option at this step so that the PDF map file is not generated. For details of **-qnoshowpdf**, see **-qshowpdf** in the *XL C Compiler Reference*.
   - Although you can specify PDF optimization (-qpdf) as early in the optimization level as **-O2**, PDF optimization is recommended at **-O4** and higher.
   - You do not have to compile all of the files of the programs with the **-qpdf1** option. In a large application, you can concentrate on those areas of the code that can benefit most from the optimization.
   - When option **-O4**, **-O5**, or any level of option **-qipa** is in effect, and you specify the **-qpdf1** option at the link step but not at the compile step, the compiler issues a warning message. The message indicates that you must recompile your program to get all the profiling information.

   Restriction: When you run an application that is compiled with **-qpdf1**, you must end the application using normal methods, including reaching the end of the execution for the main function and calling the exit() function in libc (stdlib.h) for C programs. System calls `exit()`, `_Exit()`, and `abort()` are considered abnormal termination methods and are not supported. Using abnormal program termination might result in incomplete instrumentation data generated by using the PDF file or PDF data not being generated at all.

2. Run the resulting application with a typical data set. When the application exits, profile information is written to one or more PDF files. You can train the resulting application multiple times with different data sets. The profiling information is accumulated to provide a count of how often branches are taken and blocks of code are run, based on the input data used. This step is called the PDF training step. By default, the PDF file is named `._pdf`, and it is placed in the current working directory or the directory specified by the PDFDIR environment variable. If the PDFDIR environment variable is set but the specified directory does not exist, the compiler issues a warning message. To override the defaults, use the **-qpdf1=pdfname** or **-qpdf1=exename** option.

   If you recompile your program with any **-qpdf1** option, the compiler removes the existing PDF file or files whose names and locations are the same as the file or files that will be created in the training step before generating a new application.

   Notes:

   - When you compile your program with the **-qpdf1** or **-qpdf2** option, by default, the **-qipa** option is also invoked with **level=0**.
   - To avoid wasting compile and run time, make sure that the PDFDIR environment variable is set to an absolute path. Otherwise, you might run the application from a wrong directory, and the compiler cannot locate the profiling information files. When it happens, the program might not be optimized correctly or might be stopped by a segmentation fault. A segmentation fault might also happen if you change the value of the PDFDIR environment variable and run the application before the PDF process finishes.
   - Avoid using a typical data that can distort the analysis of infrequently executed code paths.
3. If you have several PDF files, use the **mergepdf** utility to combine these PDF files into one PDF file. For example, if you produce three PDF files that represent usage patterns that occur 53%, 32%, and 15% of the time respectively, you can use this command:

   ```
   mergepdf -r 53 file_path1  -r 32 file_path2  -r 15 file_path3 -o file_path4
   ```

   where *file_path1*, *file_path2*, and *file_path3* specify the directories and names of the PDF files that are to be merged, and *file_path4* specifies the directory and name of the output PDF file.

   Notes:

   - Avoid mixing the PDF files created by different versions or PTF levels of the XL C compiler.
   - You cannot edit PDF files that are generated by the resulting application. Otherwise, the performance or function of the generated executable application might be affected.
4. 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.

   It is recommended that you use the **-qpdf2** option to link the object files that are created during the **-qpdf1** phase without recompiling your program. Using this approach, you can save considerable compilation time and achieve the same optimization result as if you had recompiled your program during the **-qpdf2** phase.

Notes:

- If the compiler cannot read any PDF files in this step, the compiler issues error message 1586-401 but continues the compilation. If you want the compiler to stop the compilation, specify **-qhaltonmsg=1586-401**.
- You are highly recommended to use the same optimization level at all compilation steps for a particular program. Otherwise, the PDF process cannot optimize your program correctly and might even slow it down. All compiler settings that affect optimization must be the same, including any supplied by configuration files.
- You can modify your source code and use the **-qpdf1** and **-qpdf2** options to compile your program. Old profiling information can still be preserved and used during the second stage of the PDF process. The compiler issues a list of warnings but the compilation does not stop. An information message is also issued with a number in the range of 0 - 100 to indicate how outdated the old profiling information is.
- When option **-O4**, **-O5**, or any level of option **-qipa** is in effect, and you specify the **-qpdf2** option at the link step but not at the compile step, the compiler issues a warning message. The message indicates that you must recompile your program to get all the profiling information.
- When using the **-qreport** option with the **-qpdf2** option, you can get additional information in your listing file to help you tune your program. This information is written to the `PDF Report` section.

5. If you want to erase the PDF information, use the **cleanpdf** utility.

Examples

The following example demonstrates that you can concentrate on compiling with **-qpdf1** only the code that can benefit most from the optimization, instead of compiling all the code with the **-qpdf1** option:

```
#Set the PDFDIR variable
export PDFDIR=$HOME/project_dir

#Compile most of the files with -qpdf1
xlc -qpdf1 -O3 -c file1.c file2.c file3.c

#This file does not need optimization
xlc -c file4.c

#Non-PDF object files such as file4.o can be linked
xlc -qpdf1 -O3 file1.o file2.o file3.o file4.o

#Run several times with different input data
./a.out < polar_orbit.data
./a.out < elliptical_orbit.data
./a.out < geosynchronous_orbit.data

#Link all the object files into the final application
xlc -qpdf2 -O3 file1.o file2.o file3.o file4.o
```

The following example bypasses recompiling the source with the **-qpdf2** option:

```
#Compile source with -qpdf1
xlc -c -qpdf1 -O3 file1.c file2.c

#Link object files
xlc -qpdf1 -O3 file1.o file2.o

#Run with one set of input data
```

```
./a.out < sample.data

#Link object files
xlc -qpdf2 -O3 file1.o file2.o
```

Related information in the *XL C Compiler Reference*

-qpdf1, -qpdf2

-O, -qoptimize

Runtime environment variables

Viewing profiling information with showpdf

With the **showpdf** utility, you can view the following types of profiling information that is gathered from your application:

- Block-counter profiling
- Call-counter profiling
- Value profiling
- Cache-miss profiling, if you specified the **-qpdf1=level=2** option during the **-qpdf1** phase.

You can view the first two types of profiling information in either text or XML format. However, you can view value profiling and cache-miss profiling information only in XML format.

Syntax

```
►►──showpdf──┬────────┬──┬─────────────┬──┬───────────────┬──┬──────┬──────────►◄
             └─pdfdir─┘  └─-f──pdfname─┘  └─-m──pdfmapdir─┘  └─-xml─┘
```

Parameters

pdfdir
Is the directory that contains the profile-directed feedback (PDF) file. If the PDFDIR environment variable is not changed after the **-qpdf1** phase, the PDF map file is also contained in this directory. If this parameter is not specified, the compiler uses the value of the PDFDIR environment variable as the name of the directory.

pdfname
Is the name of the PDF file. If this parameter is not specified, the compiler uses `._pdf` as the name of the PDF file.

pdfmapdir
Is the directory that contains the PDF map file. If this parameter is not specified, the compiler uses the value of the PDFDIR environment variable as the name of the directory.

-xml
Determines the display format of the PDF information. If this parameter is specified, the PDF information is displayed in XML format; otherwise, it is displayed in text format. Because value profiling and cache-miss profiling information can be displayed only in XML format, the PDF report in XML format contains more information than the report in text format.

Usage

A PDF map file that contains static information is generated during the **-qpdf1** phase, and a PDF file is generated during the execution of the resulting application. The **showpdf** utility needs both the PDF and PDF map files to display PDF information in either text or XML format.

By default, the PDF file is named `._pdf`, and the PDF map file is named `._pdf_map`. If the PDFDIR environment variable is set, the compiler places the PDF and PDF map files in the directory specified by PDFDIR. Otherwise, the compiler places these files in the current working directory. If the PDFDIR environment variable is set but the specified directory does not exist, the compiler issues a warning message. To override the defaults, use the **-qpdf1=pdfname** option to specify the paths and names for the PDF and PDF map files. For example, if you specify the **-qpdf1=pdfname=/home/joe/func** option, the resulting PDF file is named `func`, and the PDF map file is named `func_map`. Both of the files are placed in the `/home/joe` directory.

If the PDFDIR environment variable is changed between the **-qpdf1** phase and the execution of the resulting application, the PDF and PDF map files are generated in separate directories. In this case, you must specify the directories for both of these files to the **showpdf** utility.

Notes:
- PDF and PDF map files must be generated from the same compilation instance. Otherwise, the compiler issues an error.
- PDF and PDF map files must be generated during the same profiling process. This means that you cannot mix and match PDF and PDF map files that are generated from different profiling processes.
- You must use the same version and PTF level of the compiler to generate the PDF file and the PDF map file.
- The **showpdf** utility accepts only PDF files that are in binary format.
- You can use the PDF_WL_ID environment variable to distinguish the multiple sets of PDF counters that are generated by multiple training runs of the user program.

The following example shows how to use the **showpdf** utility to view the profiling information 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 -O hello.c
   ```
2. Run the resulting executable program **a.out** using a typical data set or several typical data sets.

3. If you want to view the profiling information for the executable file in text format, run the **showpdf** utility without any parameters.

   ```
   showpdf
   ```

   The result is as follows:

   ```
   HelloWorld(67):  1 (hello.c)

   Call Counters:
   4 | 1  printf(69)

   Call coverage = 100% ( 1/1 )

   Block Counters:
   2-4 | 1
   5 |
   5 | 1

   Block coverage = 100% ( 2/2 )

   -----------------------------------
   main(68):  1 (hello.c)

   Call Counters:
   8 | 1  HelloWorld(67)

   Call coverage = 100% ( 1/1 )

   Block Counters:
   6-9 | 1
   10 |

   Block coverage = 100% ( 1/1 )

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

   If you want to view the profiling information in XML format, run the **showpdf** utility with the **-xml** parameter.

   ```
   showpdf -xml
   ```

   The result is as follows:

   ```
     <?xml version="1.0" encoding="UTF-8" ?>
   - <XLTransformationReport xmlns="http://www.ibm.com/2010/04/CompilerTransformation" version="1.0">
     - <CompilationStep name="showpdf">
       - <ProgramHierarchy>
         - <FileList>
           - <File id="1" name="hello.c">
             - <RegionList>
                 <Region id="67" name="HelloWorld" startLineNumber="2" />
                 <Region id="68" name="main" startLineNumber="6" />
               </RegionList>
             </File>
           </FileList>
         </ProgramHierarchy>
         <TransformationHierarchy />
       - <ProfilingReports>
         - <BlockCounterList>
           - <BlockCounter regionId="67" execCount="1" coveredBlock="2" totalBlock="2">
             - <BlockList>
                 <Block index="3" execCount="1" startLineNumber="2" endLineNumber="4" />
                 <Block index="2" execCount="0" startLineNumber="5" endLineNumber="5" />
                 <Block index="4" execCount="1" startLineNumber="5" endLineNumber="5" />
               </BlockList>
             </BlockCounter>
           - <BlockCounter regionId="68" execCount="1" coveredBlock="1" totalBlock="1">
             - <BlockList>
                 <Block index="3" execCount="1" startLineNumber="6" endLineNumber="9" />
                 <Block index="2" execCount="0" startLineNumber="10" endLineNumber="10" />
               </BlockList>
             </BlockCounter>
           </BlockCounterList>
         - <CallCounterList>
   ```

```
      - <CallCounter regionId="67" execCount="1" coveredCall="0" totalCall="0">
        - <CallList>
            <Call name="printf" execCount="1" lineNumber="4" />
          </CallList>
        </CallCounter>
      - <CallCounter regionId="68" execCount="1" coveredCall="0" totalCall="0">
        - <CallList>
            <Call name="HelloWorld" execCount="1" lineNumber="8" />
          </CallList>
        </CallCounter>
      </CallCounterList>
      <ValueProfileList />
      <CacheMissList />
    </ProfilingReports>
  </CompilationStep>
</XLTransformationReport>
```

Related information in the *XL C Compiler Reference*

-qpdf1, -qpdf2

-qshowpdf

Object level profile-directed feedback

About this task

In addition to optimizing entire executables, profile-directed feedback (PDF) can also be applied to specific object files. 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 the process of relinking the entire application. In large applications, you can save the time and trouble that otherwise need to be 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 program using the **-qpdf1** option, execute the resulting application with representative data, compile the program again with the **-qpdf2** option, but now also use the **-qnoipa** option so that the linking step is skipped.

The steps below outline this process:

1. Compile your program using the **-qpdf1** option. For example:
   ```
   xlc -c -O3 -qpdf1 file1.c file2.c file3.c
   ```
   In this example, we are using the optimization level **-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
   ```
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 program again using the **-qpdf2** option. 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.
   ```
   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 by using the system loader **ld** or by omitting the **-c** option in the **-qpdf2** step.

Notes:

- You must use the same optimization level in all the steps. In this example, the optimization level is **-O3**.
- 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 is `._pdf`; the location of the file is the current working directory or whatever directory you have set using the PDFDIR environment variable. If the PDFDIR environment variable is set but the specified directory does not exist, the compiler issues a warning message.
- Because the **-qnoipa** option needs to be specified in the **-qpdf2** step so that linking of your object files is skipped, you cannot use interprocedural analysis (IPA) optimizations and object level PDF at the same time.

For details, see -qpdf1, -qpdf2 in the XL C Compiler Reference.

Handling table of contents (TOC) overflow

To handle table of contents (TOC) overflow, you can reduce the number of global symbols, enlarge the TOC access range, or apply interprocedural analysis.

The addresses of global symbols in programs are stored in a data structure called TOC. To access a global symbol, the address of the global symbol must be retrieved from the TOC. The default TOC data structure has a fixed size that can store a fixed number of global symbols. For example, the IBM PowerPC architecture uses an instruction with a signed 16-bit offset for indirect address calculations and limits the size of the TOC to 64 KB. A maximum of 16 K entries can be stored in the TOC in 32-bit mode and 8 K entries in 64-bit mode.

For large applications, it is common to have more global symbols than can be stored in the default TOC. If an application contains more TOC entries than the TOC can hold, the linker reports TOC overflow, indicating that an alternative mechanism must be used. Use the following approaches to handle TOC overflow:

- Reduce the number of global symbols in programs in the following ways:
  - Change the source code. It is the best approach to reduce the number of global symbols.
  - Specify the **-qminimaltoc** option.
  - Apply interprocedural analysis by specifying the **-qipa** option. For more information about the option, see "Using interprocedural analysis" on page 55 and "Getting the most from -qipa" on page 56.
- Enlarge the TOC access range by specifying the following options:
  - **-bbigtoc** (a linker option)
  - **-qpic=large**

Options for reducing the number of global symbols

The best approach to handle table of contents (TOC) overflow is to reduce the number of global symbols so that the required number of TOC entries is also reduced.

You can reduce the number of global symbols by refining the source code. If possible, change the source code to remove unnecessary global variables and functions, mark them as static, or group global variables in structures. However, changing the source code can be time consuming and error prone. In fact, the compiler can finish these tasks automatically when link-time optimization is used at high optimization levels (**-O4** and **-O5**) or the optimization is applied explicitly with **-qipa** at both compile and link time. The optimization result is similar to what can be achieved through source changes but without widespread manual source changes.

To reduce the number of global symbols, you can also specify the **-qminimaltoc** option. When you specify this option, the compiler creates a separate table for each source file. The table contains the address of each global symbol that is used in the source file. Specifying the option ensures that the compiler creates only one TOC entry for each compilation unit. This option is effective only if the TOC entries are spread across multiple source files. If a single source file contains enough global symbols to cause TOC overflow, the option has no effect to help with TOC overflow.

Notes:

- It is unnecessary to specify **-qminimaltoc** for each compilation unit.
- Use **-qminimaltoc** with discretion because it might lead to low performance. The use of **-qminimaltoc** introduces indirect reference and hence increases the time that is required to access a global symbol. Another drawback is that the memory requirements for the application might increase. For more information about the performance impact of the option, see “Performance considerations of handling TOC overflow” on page 67.

Options for enlarging the TOC access range

Enlarging the table of contents (TOC) access range is an effective way to handle TOC overflow. Two instructions are used to access the TOC and group the ranges to consecutive TOC regions. The maximum 16-bit offset on IBM PowerPC supports a large TOC of 64 K TOC regions. With a maximum of 64 K entries in each TOC region, a large TOC can be 4 GB. It creates a limit of 1 G global symbols in a 32-bit environment and 500 M in a 64-bit environment. On POWER8 systems, the two instructions are often executed together in the same time as one.

To enlarge a TOC, you can specify the **-bbigtoc** or **-qpic=large** option. Before you specify the options, reduce the number of TOC entries because a program that contains generated code can result in poor performance.

-bbigtoc

The **-bbigtoc** option is a linker option of the **-b** flag. It generates extra code if the size of TOC is greater than 64 K. To increase the total TOC capacity, extended TOC regions are created in addition to the base TOC. As a result, the base TOC is the first 64 K region, followed by one or more 64 K regions that form the extended TOC. When the address of a global symbol is placed in the extended TOC, the linker inserts a branch to out-of-line code that contains instructions to compute the location within the extended TOC. The location of the symbol is computed with three instructions: one to locate the extended TOC region, the second to compute the location within the extended TOC, and the third to branch back. When you specify this option, the execution time is increased.

Note: The **-bbigtoc** option is a linker option and the code that is produced by the compiler is not changed when the option is specified.

-qpic=large

The **-qpic=large** option works with the linker to generate more efficient code than the **-bbigtoc** option does. When **-qpic=large** is specified, the compiler always generates two instructions to get the address of a symbol, whether TOC overflow occurs or not. When the option is specified, all the symbols, including the symbols in the base TOC, require an extra instruction to compute the addresses. For offsets within the normal 64 KB base TOC size, the linker converts the first instruction to one that does nothing and takes no execution time. On POWER8, these two instructions are often merged into one instruction with a larger displacement. The linker does not insert a branch to out-of-line code for offsets. For more information about how performance is affected by the option, see "Performance considerations of handling TOC overflow."

Note: It is unnecessary to specify **-qpic=large** for each compilation unit.

Performance considerations of handling TOC overflow

Performance must be considered when you handle table of contents (TOC) overflow. When you bypass TOC overflow, minimize any negative effects on runtime performance.

When TOC overflow occurs, the best solution is to reduce the number of global symbols by modifying the source code. You can also consider specifying the following options, depending on performance requirements:

-qipa

- **Advantage:** Applying the interprocedural analysis (IPA) process significantly reduces TOC pressure. In many cases, it completely eliminates TOC overflow. IPA does so by restructuring your program so that the number of global symbols is reduced.
- **Consideration:** IPA is implied at optimization levels **-O4** and **-O5**, but those also include other complex optimizations that are not relevant to commercial application development. One good alternative is **-qipa=level=0**, which applies a minimal level of whole program optimization; however, you might need **-qipa=level=1** for large applications. Level 1 runs a more aggressive reduction of the TOC requirements, at the cost of a longer compilation process.

Note: For whole program analysis, you must specify the **-qipa** option at both the compile and link command lines.

-qminimaltoc

- **Advantage:** If the source code contains only performance insensitive global symbols, use **-qminimaltoc** to compile the file. The option places all global symbols in a source file into a separate data structure, which is useful to reduce overall pressure on the TOC.
- **Consideration:** You must specify the option with discretion, particularly with files that contain frequently executed code. The use of the option can significantly affect performance when it is used to compile files that contain performance-sensitive global symbols. When **-qminimaltoc** is specified, the program is larger and slower than it was. However, it is still faster than using the **-bbigtoc** option.

Note: You do not need to specify **-qminimaltoc** on all compilation units. Minimize the negative performance by using this option only on compilation units that are not performance relevant.

-bbigtoc

- **Advantage:** When you specify the option, there is no additional performance cost to access global symbols that are stored in the base TOC.
- **Consideration:**If you specify the **-bbigtoc** option, accessing global symbols in the extended TOC requires more instructions to be executed, including a branch to out-of-line code. The execution time for the sequence of instructions is much more than for the two instructions that are generated by **-qpic=large**, although there is no increased execution time to load pointers within the normal 64 KB base TOC size. In addition, the compiler can miss opportunities to optimize the file because handling the offset calculation is generated at link time.

-qpic=large

- **Advantage:** The **-qpic=large** option is usually the preferred solution, because it provides the best balance for accessing symbols in both the base and the extended TOC.
- **Consideration:** The option might affect performance because it uses two instructions to get the address of a symbol, regardless of whether TOC overflow occurs. However, the two instructions have only a short latency when compared with the sequence generated by **-bbigtoc**. On POWER8 based system, the two instructions are often executed together in the same time as one.

Marking variables as local or imported

The compiler assumes that all variables in applications are imported, but the use of -qdatalocal and -qdataimported can mark variables local or imported. The compiler optimizes applications that are based on the specification of static or dynamic binding for program variables.

-qdatalocal

Local variables are stored in a special segment of memory that is uniquely bound to a program or shared library. Specify the -qdatalocal option to identify variables to be treated as local to a compiled program or shared library. You can specify the option with no parameters to indicate that all appropriate variables are local. Alternatively, you can append a list of colon-separated names to the option to treat only a subset of the program arguments as local.

When it can be, a variable that is marked as local is embedded directly into a structure that is called the table of contents (TOC) instead of in a separate global piece of memory. The prerequisite is that the variable's storage must be no more than the pointer size for it to be embedded in the TOC. Usually, pointers to data are stored in the TOC. The -qdatalocal option allows storage of data directly in the TOC, hence reducing data accesses from two load instructions to one load instruction.

-qdataimported

Imported variables are stored according to the default memory allocation scheme. The -qdataimported option is the default data binding mechanism. Specifying the option implies that the data is visible to other program or shared library that is linked. As a result, specifying variable names as arguments to the `-qdataimported` option or compiling with the `-qdataimported` option without arguments in isolation has no effect.

The `-qdataimported` option is useful when you use it in combination with `-qdatalocal`. Because it is unlikely that you want to store all data in the TOC, the `-qdataimported` option can override `-qdatalocal` for variables external to a program or shared library. For example, the use of options `-qdatalocal -qdataimported=<variable>` stores all global data in the TOC except for <*variable*>.

Related information in the *XL C Compiler Reference*

-qdataimported, -qdatalocal, -qtocdata

Getting the most out of -qdatalocal

You can see some examples that illustrate the use of the `-qdatalocal` option.

In the source for the following program file, `A1` and `A2` are global variables:

```
int A1;
int A2;
int main(){
  A2=A1+1;
  return A2;
}
```

Here is an excerpt of the listing file that is created if you specify `-qlist` without `-qdatalocal`:

```
 | 000000                              PDEF     main
4|                                     PROC
5| 000000 lwz      80620004   1        L4A       gr3=.A1(gr2,0)
5| 000004 lwz      80630000   1        L4A       gr3=A1(gr3,0)
5| 000008 addi     38630001   1        AI        gr3=gr3,1
5| 00000C lwz      80820008   1        L4A       gr4=.A2(gr2,0)
5| 000010 stw      90640000   1        ST4A      A2(gr4,0)=gr3
```

Here is an excerpt of the listing file that is created if you specify `-qlist` with `-qdatalocal`:

```
 | 000000                              PDEF     main
4|                                     PROC
5| 000000 lwz      80620004   1        L4A       gr3=A1(gr2,0)
5| 000004 addi     38630001   1        AI        gr3=gr3,1
5| 000008 stw      90620008   1        ST4A      A2(gr2,0)=gr3
```

When you specify `-qdatalocal`, the data is accessed by a single load instruction because the `A1` and `A2` variables are embedded in the TOC. When you do not specify `-qdatalocal`, `A1` and `A2` variables are accessed by two load instructions. In this example, you can use `-qdatalocal=A1:A2` to specify local variables individually.

You can always see the section that begins with `>>>>> OPTIONS SECTION <<<<<` in the `.lst` file that is created by `-qlist` to confirm the use of these options. For example, you can view `DATALOCAL=<variables>` or `DATALOCAL` when the option is specified.

Notes:

- On AIX, TOC entries are pointer size. When you specify `-qdatalocal` without arguments, the option is ignored for variables that are larger than the pointer size. Conversely, data smaller than pointer size is word-aligned. See the following example of an `objdump` excerpt that shows when a `char (r3)` is marked local. The offset between the byte and the next data `(r4)` is still 4 bytes. The data is accessed by a load byte instruction instead of a regular load. For more information about how TOC stores data, see Handling table of contents (TOC) overflow.

```
10000380:       88 62 00 20     lbz     r3,32(r2)
10000384:       80 82 00 24     l       r4,36(r2)
r2 (base address of the TOC), r3 (char), r4 (int)
```

- If you specify an unsuitable variable as a parameter to `-qdatalocal`, `-qdatalocal` is ignored. Unsuitable variables can be data that exceeds pointer-size bytes or variables that do not exist. When you specify `-qdatalocal` for a variable that is not a TOC candidate, the default storage for that variable is set to `-qdataimported` and the variable is not stored in the TOC.
-
- Mark variables as local with care. If you specify `-qdatalocal` without any arguments, expect all global variables to be candidates for TOC direct placement, even those variables that are marked as external. Variables with static linkage do not have the same issues.
- Since each TOC structure is unique to a module or shared library, the utility of the `-qdatalocal` option is limited to data within that module or shared library.
- For programs with multiple modules, switching between multiple TOC structures might dilute the speedup that is associated with this option.

Related information in the *XL C Compiler Reference*

-qdataimported, -qdatalocal, -qtocdata

Using compiler reports to diagnose optimization opportunities

You can use the **`-qlistfmt`** option to generate a compiler report in XML or HTML format. It provides information about how your program is optimized. You can also use the **`genhtml`** utility to convert an existing XML report to HTML format. This information helps you understand your application codes and tune codes for better performance.

The compiler report in XML format can be viewed in a browser that supports XSLT. If you compile with the **`stylesheet`** suboption, for example, **`-qlistfmt=xml=all:stylesheet=xlstyle.xsl`**, the report contains a link to a stylesheet that renders the XML readable. By reading the report, you can detect opportunities to further optimize your code. You can also create tools to parse this information.

By default, the name of the report is `a.xml` for XML format, and `a.html` for HTML format. You can use the **`-qlistfmt=xml=filename`** or **`-qlistfmt=html=filename`** option to override the default name.

Inline reports

If you compile with **`-qinline`** and one of **`-qlistfmt=xml=inlines`**, **`-qlistfmt=html=inlines`**, **`-qlistfmt=xml`**, or **`-qlistfmt=html`**, the generated compiler report includes a list of inline attempts during compilation. The report also specifies the type of attempt and its outcome.

For each function that the compiler has attempted to inline, there is an indication of whether the inline was successful. The report might contain any number of reasons why a named function has not been successfully inlined. Some examples of these reasons are as follows:

- FunctionTooBig - The function is too big to be inlined.
- RecursiveCall - The function is not inlined because it is recursive.
- ProhibitedByUser - Inlining was not performed because of a user-specified pragma or directive.
- CallerIsNoopt - No inlining was performed because the caller was compiled without optimization.
- WeakAndNotExplicitlyInline - The calling function is weak and not marked as inline.

For a complete list of the possible reasons, see the `Inline optimization types` section of the XML schema help file named `XMLContent.html` in the `/opt/IBM/xlc/13.1.2/listings/` directory. The Japanese and Chinese versions of the help file, `XMLContent-Japanese.utf8.html` and `XMLContent-Chinese.utf8.html`, are included in this directory as well.

Loop transformations

If you compile with **`-qhot`** and one of **`-qlistfmt=xml=transforms`**, **`-qlistfmt=html=transforms`**, **`-qlistfmt=xml`** or **`-qlistfmt=html`**, the generated compiler report includes a list of the transformations performed on all loops in the file during compilation. The report also lists the reasons why transformations were not performed in some cases:

- Reasons why a loop cannot be automatically parallelized
- Reasons why a loop cannot be unrolled
- Reasons why SIMD vectorization failed

For a complete list of the possible transformation problems, see the `Loop transformation types` section of the XML schema help file named `XMLContent.html` in the `/opt/IBM/xlc/13.1.2/listings/` directory.

Data reorganizations

If you compile with **`-qhot`** and one of **`-qlistfmt=xml=data`**, **`-qlistfmt=html=data`**, **`-qlistfmt=xml`**, or **`-qlistfmt=html`**, the generated compiler report includes a list of data reorganizations performed on the program during compilation. Here are some examples of data reorganizations:

- Array splitting
- Array coalescing
- Array interleaving
- Array transposition
- Memory merge

For each of these reorganizations, the report contains details about the name of the data, file names, line numbers, and the region names.

Profile-directed feedback reports

If you compile with **-qpdf2** and one of **-qlistfmt=xml=pdf**, **-qlistfmt=html=pdf**, **-qlistfmt=xml**, or **-qlistfmt=html**, the generated compiler report includes the following information:

- Loop iteration counts
- Block and call counts
- Cache misses (if compiled with **-qpdf1=level=2**)

Related information

-qlistfmt

Parsing compiler reports with development tools

You can write tools to parse the compiler reports produced in XML format to help you find opportunities to improve application performance.

The compiler includes an XML schema that you can use to create a tool to parse the compiler reports and display aspects of your code that might represent performance improvement opportunities. The schema, `xllisting.xsd`, is located in the `/opt/IBM/xlc/13.1.2/listings/` directory. This schema helps present the information from the report in a tree structure.

You can also find a schema help file named `XMLContent.html` that helps you understand the schema details. The Japanese and Chinese versions of the help file, `XMLContent-Japanese.utf8.html` and `XMLContent-Chinese.utf8.html`, are in the same directory.

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 about these options, see the heading for each option in the *XL C Compiler Reference*.

Table 23. 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.
-qsmallstack	Instructs the compiler to compact stack storage. Doing so might increase heap usage, which might increase execution time. However, it might be necessary for the program to run or to be optimally multithreaded.
-qinline	Controls inlining.
-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.

Table 23. Selected compiler options for optimizing performance (continued)

Option	Description
-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 nonvolatile register saves and restores. It should not be used if the program uses `setjmp/longjmp` or any other form of exception handling.
-qstrict	Disables all transformations that change program semantics. In general, compiling a correct program with **-qstrict** and any levels of optimization produces the same results as without optimization.
-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 the **-O3** and higher optimization levels.
-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.
-qprefetch	Inserts prefetch instructions in compiled code to improve code performance. In situations where you are working with applications that generate a high cache-miss rate, you can use its suboption **assistthread** to generate prefetching assist threads (for example, `-qprefetch=assistthread`). **-qnoprefetch** is the default option.

Related information in the *XL C Compiler Reference*

- -qignerrno
- -qsmallstack
- -qinline
- -qunroll / #pragma unroll
- -qinlglue
- -qtbtable
- -qunwind
- -qstrict
- -qlargepage
- -qprefetch

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 might not be current if the most current value is being stored in a register. Most debuggers cannot follow the removal of stores to a variable, and to the debugger it appears as though that variable is never updated, or possibly even never 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 might reorder instructions. That is, instructions might not be executed in the order you would expect based on the sequence of lines in the original source code. Also, the sequence of instructions for a statement might not be contiguous. As you step through the program with a debugger, the program might appear as if it is returning to a previously executed line in the 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 might 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, and then recompile it with your desired optimization options. See "Debugging in the presence of optimization" on page 76 for some compiler options that are useful in this approach.

Use -g level
Use the **-g** level suboption to control the amount of debugging information made available. Increasing it improves debug capability but prevents some optimizations. For more information, see **-g**.

Use -qoptdebug
When compiling with **-O3** optimization level 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. For more information, see "Using -qoptdebug to help debug optimized programs" on page 77.

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***) might 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 such programs might fail when you optimize. Similarly, a program can appear to execute correctly after optimization, but it fails at lower optimization levels or when it is run in a different environment. You can also use the **-qcheck=unset** option and **-qinfo=unset** option to detect variables that are not or might not be initialized.
- 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, unless the **-g8** or **-g9** option is in effect and the optimization level is **-O2**. 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 in the presence of optimization

Debug and compile your program with your desired optimization options. Test the optimized program before placing it 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 debugging of optimized code:

-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, how the program was parallelized, what inlining was done, and some other transformations. To make **-qreport** generate a listing file, you must also specify one of the **-qhot**, **-qsmp**, **-qinline**, **-qsimd**, or other optimization options.

-qinfo=mt
: Reports potential synchronization issues in parallel code. For details, see -qinfo.

-qinfo=unset
: Detects automatic variables that are used before they are set, and flags them with informational messages at compile time. For details, see -qinfo.

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

-qcheck
: Generates code that performs certain types of runtime checking.

-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.

-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.

-g
: Generates debugging information to be used by a symbolic debugger. You can use different **`-g`** levels to debug optimized code by viewing or possibly modifying accessible variables at selected source locations in the debugger. Higher **`-g`** levels provide a more complete debug support, while lower levels provide higher runtime performance. For details, see **`-g`**.

In addition, you can also use the **snapshot** pragma to ensure that certain variables are visible to the debugger at points in your application. For details, see **#pragma ibm snapshot**.

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 **`-qoptdebug`** 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 are generated with file suffix `.optdbg`. Only line debugging is supported for this feature.

Note: The compiler has introduced support for **`-g`** to provide support for various levels of trade-off between full debug support and optimization. If you want to debug your source code while taking advantage of compiler optimizations, use **-g** instead of **-qoptdebug**. For more information, see **-g**.

Compile your program as in the following example:

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

In this example, your source file is compiled to `a.out`. The pseudocode for the optimized program is written to a file called `myprogram.optdbg`, which can be referred to when you debug 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 is specified, the pseudocode file `<output_file>.optdbg` that contains the optimized pseudocode is still generated.
- The **-qoptdebug** option takes effect only when one or more of the optimization options **-qhot**, **-qsmp**, **-qpdf**, or **-qipa** 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

From the following examples, you can see how the compiler might apply optimizations to a simple program and how debugging it can differ from debugging your original source.

Example 1: 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 can see iterations of the loop listed explicitly.

Example 2: 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.

Example 3: Shows an example of stepping through the optimized source using the debugger. Note that there is no longer a correspondence between the line numbers for these statements in the optimized source as compared with the line numbers in the original source.

Example 1: Original code

```
#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;
}
```

Example 2: dbx debugger listing

```
(dbx) list
    1       3 |  void foo(long x, long y, char * w)
    2       4 |  {
    3       9 |    a = printf("TEST\n");
    4      12 |    printf("%d %d %d %s %s\n",a,x + y,x + y,
                          ((char *)w  + 1),((char *)w  + 1));
    5              printf("%d %d %d %s %s\n",a,x + y,x + y,
                          ((char *)w  + 1),((char *)w  + 1));
    6              printf("%d %d %d %s %s\n",a,x + y,x + y,
                          ((char *)w  + 1),((char *)w  + 1));
    7              printf("%d %d %d %s %s\n",a,x + y,x + y,
                          ((char *)w  + 1),((char *)w  + 1));
    8      13 |    return;
    9            } /* function */
   10
   11
   12      15 |  long main()
   13      16 |  {
   14      17 |    d$init$0 = "DEBUG";
   15      18 |    @PARM.x0 = 3;
   16              @PARM.y1 = 4;
   17              @PARM.w2 = &d;
   18       9 |    a = printf("TEST\n");
   19      12 |    printf("%d %d %d %s %s\n",a,@PARM.x0 + @PARM.y1,
                           @PARM.x0 + @PARM.y1,((char *)@PARM.w2  + 1),
                           ((char *)@PARM.w2  + 1));
   20              printf("%d %d %d %s %s\n",a,@PARM.x0 + @PARM.y1,
                           @PARM.x0 + @PARM.y1,((char *)@PARM.w2  + 1),
                           ((char *)@PARM.w2  + 1));
   21              printf("%d %d %d %s %s\n",a,@PARM.x0 + @PARM.y1,
                           @PARM.x0 + @PARM.y1,((char *)@PARM.w2  + 1),
                           ((char *)@PARM.w2  + 1));
   22              printf("%d %d %d %s %s\n",a,@PARM.x0 + @PARM.y1,
                           @PARM.x0 + @PARM.y1,((char *)@PARM.w2  + 1),
                           ((char *)@PARM.w2  + 1));
   23      19 |    rstr = 0;
   24              return rstr;
   25      20 |  } /* function */
```

Example 3: Stepping through optimized source

```
(dbx) stop at 18
[1] stop at "myprogram.o.optdbg":18
(dbx) run
[1] stopped in main at line 18 in file "myprogram.o.optdbg"
   18       9 |    a = printf("TEST\n");
(dbx) cont
TEST
5 7 7 EBUG EBUG
5 7 7 EBUG EBUG
5 7 7 EBUG EBUG
5 7 7 EBUG EBUG

execution completed
```

Chapter 9. Coding your application to improve performance

Chapter 7, "Optimizing your applications," on page 43 discusses the various compiler options that the XL C compiler 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:

- "Finding faster input/output techniques"
- "Reducing function-call overhead"
- "Optimizing variables" on page 82
- "Manipulating strings efficiently" on page 83
- "Optimizing expressions and program logic" on page 84
- "Optimizing operations in 64-bit mode" on page 85
- "Tracing functions in your code" on page 85
- "Using visibility attributes (IBM extension)" on page 89

Finding faster input/output techniques

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

- If your file I/O accesses do not exhibit locality (that is truly random access such as in a database), implement your own buffering or caching mechanism on the low-level I/O functions.
- If you do your own I/O buffering, make the buffer a multiple of 4KB, which is the minimum size of a page.
- Use buffered I/O to handle text files.
- If 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.

Reducing 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.
- Use `const` 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 the `restrict` keyword for pointers that can never point to the same memory.
- 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 they have few parameters and the most frequently used parameters are in the leftmost positions in the function prototype.
- Avoid passing by value large structures or unions as function parameters or returning a large 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` or small aggregates 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 they often allow the compiler to perform better optimization.

  Many functions from `string.h` and `math.h` are mapped to optimized built-in functions by the compiler.
- 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 are generally not good candidates for inlining. Neither are medium size functions that are called from many places.
- 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 using 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

Optimizing variables

Consider the following guidelines:

- Use local variables, preferably automatic variables, as much as possible. The compiler must make several worst-case assumptions about global variables. For example, if a function uses external variables and also calls external functions, the compiler assumes that every call to an external function could use and change the value of every external variable. If you know that a global variable is not read or 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. Do not group variables whose addresses are taken with variables whose addresses are not taken.
- 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 for a different purpose. Taking the address of a local variable can inhibit optimizations that would otherwise be done on calculations involving that variable.
- Use constants instead of variables where possible. The optimizer is 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++)`). An enumeration declaration can be used to declare a named constant for maintainability.
- Use register-sized integers (`long` data type) for scalars to avoid sign extension instructions after each change in 64-bit mode. For large arrays of integers, consider using one-byte 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 high precision is required.

Related information in the *XL C Compiler Reference*

-qisolated_call / #pragma isolated_call

Manipulating 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 or 16-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 , or it might copy in reverse order depending on the length of the string.

- When manipulating strings using `mem` functions, faster code can 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. When the same string is used multiple times, making it read-only improves certain optimization techniques, reduces memory usage, and shortens compilation time. You can explicitly set strings to read-only by using **`#pragma strings (readonly)`** in your source files or **`-qro`** (this is enabled by default except when compiling with **cc**) to avoid changing your source files.

Related information in the *XL C Compiler Reference*

-qro / #pragma strings

Optimizing expressions and program logic

Consider the following guidelines:

- If components of an expression are used in other expressions and they include function calls or there are function calls between the uses, 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.
- Do not use global variables as loop indices or bounds.
- 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. If possible, replace the `switch` statement by checking whether the value is in range to be obtained from an array.
- Keep array index expressions as simple as possible.

Optimizing 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 or loop counts might require additional instructions to perform sign extension each time the array is referenced or the loop count is incremented.

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

- Avoid performing mixed 32-bit and 64-bit operations. For example, adding a 32-bit data type to a 64-bit data type requires that the 32-bit be sign-extended to clear or set the upper 32-bit of the register. This slows the computation.
- Use `long` types instead of `signed`, `unsigned`, and plain `int` types for variables that 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.

Tracing functions in your code

You can instruct the compiler to insert calls to user-defined tracing functions to aid in debugging or timing the execution of other functions.

Using tracing functions in your program requires the following steps:

1. Writing tracing functions
2. Specifying which functions to trace with the **-qfunctrace** option

Using the **-qfunctrace** option causes the compiler to insert calls to these tracing functions at key points in the function body; however, you are responsible for defining these tracing functions. The following list describes at which points the tracing functions are called:

- The compiler inserts calls to the tracing function at the entry point of a function. The line number passed to the routine is the line number of the first executable statement in the instrumented function.
- The compiler inserts calls to the tracing function at the exit point of a function. The line number that is passed to the function is the line number of the statement causing the exit in the instrumented function.

You can use the **-qnofunctrace** compiler option or the `#pragma nofunctrace` pragma to disable function tracing.

How to write tracing functions

To trace functions in your code, define the following tracing functions:

- `__func_trace_enter` is the entry point tracing function.
- `__func_trace_exit` is the exit point tracing function.

The prototypes of these functions are as follows:

- `void __func_trace_enter(const char *const function_name, const char *const file_name, int line_number, void **const user_data);`
- `void __func_trace_exit(const char *const function_name, const char *const file_name, int line_number, void **const user_data);`

In the preceding tracing functions, the descriptions for their variables are as follows:

- `function_name` is the name of the function you want to trace.
- `file_name` is the name of the file.
- `line_number` is the line number at entry or exit point of the function. This is a 4-byte number.
- `user_data` is the address of a static pointer variable. The static pointer variable is generated by the compiler and initialized to `NULL`; in addition, because the pointer variable is static, its address is the same for all instrumentation calls inside the same function.

Notes:

- The exit function is not called if the function has an abnormal exit. The abnormal exit can be caused by actions such as raising a signal or calling the exit() or abort() functions.
- The **`-qfunctrace`** option does not support `setjmp` and `longjmp`. For example, a call to `longjmp()` that leaves function1 and returns from `setjmp()` in function2 will have a missing call to `__func_trace_exit` in function1 and a missing a call to `__func_trace_enter` in function2.
- To define tracing functions in C++ programs, use the `extern "C"` linkage directive before your function definition.
- The function calls are only inserted into the function definition, and if a function is inlined, no tracing is done within the inlined code.
- If you develop a multithreaded program, make sure the tracing functions have the proper synchronization and do not cause deadlock. Calls to the tracing functions are not thread-safe.
- If you specify a function that does not exist with the option, the function is ignored.

Rules

The following rules apply when you trace functions in your code:

- When optimization is enabled, line numbers might not be accurate.
- The tracing function must not call any instrumented function; otherwise an infinite loop might occur.
- If you instruct the compiler to trace recursive functions, make sure that your tracing functions can handle recursion.
- Inlined functions are not instrumented.
- Tracing functions are not instrumented.
- Compiler-generated functions are not instrumented, except for the outlined functions generated by optimization such as OpenMP. In those cases, the name of the outlined function contains the name of the original user function as prefix.

Examples

The following C example shows how you can trace functions in your code using function prototypes. Assume you want to trace the entry and exit points of `function1` and `function2`, as well as how much time it takes the compiler to trace them in the following code:

Main program file: t1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#ifdef __cplusplus
extern "C"
#endif
void __func_trace_enter(const char *function_name, const char *file_name,
                        int line_number, void** const user_data){
   if((*user_data)==NULL)
       (*user_data)=(time_t *)malloc(sizeof(time_t));
   (*(time_t *)*user_data)=time(NULL);
   printf("begin function: name=%s file=%s line=%d\n",function_name,file_name,
          line_number);
}
#ifdef __cplusplus
extern "C"
#endif
void __func_trace_exit(const char *function_name, const char*file_name,
                       int line_number, void** const user_data){
   printf("end function: name=%s file=%s line=%d. It took %g seconds\n",
          function_name,file_name,line_number, difftime(time(NULL),
          *(time_t *)*user_data));
}
void function2(void){
   sleep(3);
}
void function1(void){
   sleep(5);
   function2();
}
int main(){
   function1();
}
```

Compile the main program source file as follows:

```
xlc t1.c -qfunctrace+function1:function2
```

Run executable `a.out` to output function trace results:

```
begin function: name=function1 file=t.c line=27
begin function: name=function2 file=t.c line=24
end function: name=function2 file=t.c line=25. It took 3 seconds
end function: name=function1 file=t.c line=29. It took 8 seconds
```

As you see from the preceding example, the `user_data` parameter is defined to use the system time as basis for time calculation. The following steps explain how `user_data` is defined to achieve this goal:

1. The function reserves a memory area for storing the value of `user_data`.
2. The system time is used as the value for `user_data`.
3. In the `__func_trace_exit` function, the `difftime` function uses `user_data` to calculate time differences. The result is displayed in the form of `It took %g seconds` in the output.

Main program file: t2.cpp

```
#include <iostream>
#include <vector>
#include <stdexcept>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
extern "C"
void __func_trace_enter(const char *function_name, const char *file_name,
                        int line_number, void** const user_data){
  if((*user_data)==NULL)
    (*user_data)=(time_t *)malloc(sizeof(time_t));
  (*(time_t *)*user_data)=time(NULL);
  printf("enter function: name=%s file=%s line=%d\n",function_name,file_name,
         line_number);
}
extern "C"
void __func_trace_exit(const char *function_name, const char*file_name,
                       int line_number, void** const user_data){
  printf("exit function: name=%s file=%s line=%d. It took %g seconds\n",
         function_name, file_name, line_number, difftime(time(NULL),
         *(time_t *)*user_data));
}
extern "C"
void __func_trace_catch(const char *function_name, const char*file_name,
                        int line_number, void** const user_data){
  printf("catch function: name=%s file=%s line=%d. It took %g seconds\n",
         function_name, file_name,line_number, difftime(time(NULL),
         *(time_t *)*user_data));
}

template <typename T> class myStack{
  private:
  std::vector<T> elements;
  public:
  void push(T const&);
  void pop();
};

template <typename T>
void myStack<T>::push(T const& value){
  sleep(3);
  std::cout<< "\tpush(" << value << ")" <<std::endl;
  elements.push_back(value);
}
template <typename T>
void myStack<T>::pop(){
  sleep(5);
  std::cout<< "\tpop()" <<std::endl;
  if(elements.empty()){
    throw std::out_of_range("myStack is empty");
  }
  elements.pop_back();
}
void foo(){
  myStack<int> intValues;
  myStack<float> floatValues;
  myStack<double> doubleValues;
  intValues.push(4);
  floatValues.push(5.5f);
  try{
    intValues.pop();
    floatValues.pop();
    doubleValues.pop(); // cause exception
  } catch(std::exception const& e){
    std::cout<<"\tException: "<<e.what()<<std::endl;
  }
  std::cout<<"\tdone"<<std::endl;
}
#pragma nofunctrace(main)
int main(){
  foo();
}
```

Compile the main program source file as follows:

```
xlC t2.cpp -qfunctrace+myStack:foo
```

Related information

- For details about the **-qfunctrace** compiler option, see -qfunctrace in the *XL C Compiler Reference*.
- See #pragma nofunctrace in the *XL C Compiler Reference* for details about the `#pragma nofunctrace`.

Using visibility attributes (IBM extension)

Visibility attributes describe whether and how an entity that is defined in one module can be referenced or used in other modules. Visibility attributes affect entities with external linkage only, and they cannot increase the visibility of other entities. By specifying visibility attributes for entities, you can export only the entities that are necessary to shared libraries. With this feature, you can get the following benefits:

- Decrease the size of shared libraries.
- Reduce the possibility of symbol collision.
- Allow more optimization for the compile and link phases.
- Improve the efficiency of dynamic linking.

Supported types of entities

The compiler supports visibility attributes for the following entities:

- Function
- Variable

Note: Data types in the C language do not have external linkage, so you cannot specify visibility attributes for C data types.

Related information in the *XL C Compiler Reference*

-qvisibility (IBM extension)

-G

-qmkshrobj

#pragma GCC visibility push, #pragma GCC visibility pop (IBM extension)

Related information in the *XL C Language Reference*

The visibility variable attribute (IBM extension)

The visibility function attribute (IBM extension)

Internal linkage

External linkage

No linkage

Types of visibility attributes

The following table describes different visibility attributes.

Table 24. Visibility attributes

Attribute	**Description**
default	Indicates that external linkage entities have the default attribute in object files. These entities are exported in shared libraries, and can be preempted.
protected	Indicates that external linkage entities have the protected attribute in object files. These entities are exported in shared libraries, but cannot be preempted.
hidden	Indicates that external linkage entities have the hidden attribute in object files. These entities are not exported in shared libraries, but their addresses can be referenced indirectly through pointers.
internal	Indicates that external linkage entities have the internal attribute in object files. These entities are not exported in shared libraries, and their addresses are not available to other modules in shared libraries.

Notes:

- In this release, the hidden and internal visibility attributes are the same. The addresses of the entities that are specified with either of these visibility attributes can be referenced indirectly through pointers.
- On the AIX platform, all the external linkage entities do not have visibility attributes by default if they do not get visibility attributes from the compiler option, pragma directives, explicitly specified attributes, or propagation rules. Whether these entities are exported in shared libraries depends on the specified export list or the one that is generated by the compiler.
- On the AIX platform, entity preemption occurs only when runtime linking is used. For details, see "Linking a library to an application" in the *XL C Optimization and Programming Guide* .

Example: Differences among the default, protected, hidden, and internal visibility attributes

```
//a.c
#include <stdio.h>
void __attribute__((visibility("default"))) func1(){
   printf("func1 in the shared library");
}
void __attribute__((visibility("protected"))) func2(){
   printf("func2 in the shared library");
}
void __attribute__((visibility("hidden"))) func3(){
   printf("func3 in the shared library");
}
void __attribute__((visibility("internal"))) func4(){
   printf("func4 in the shared library");
}

//a.h
extern void func1();
extern void func2();
extern void func3();
extern void func4();

//b.c
#include "a.h"
void temp(){
   func1();
   func2();
}

//b.h
extern void temp();

//main.c
#include "a.h"
#include "b.h"

void func1(){
   printf("func1 in b.c");
}
void func2(){
   printf("func2 in b.c");
}
void main(){
   temp();
   // func3(); // error
   // func4(); // error
}
```

You can use the following commands to create a shared library named `libtest.so`:

```
xlc -c -qpic a.c b.c
xlc -G -o libtest.so a.o b.o
```

Then, you can dynamically link `libtest.so` during run time by using the following commands:

```
xlc main.c -L. -ltest -brtl -bexpall -o main
./main
```

The output of the example is as follows:

```
func1 in b.c
func2 in the shared library
```

The visibility attribute of function `func1()` is default, so it is preempted by the function with the same name in `main.c`. The visibility attribute of function `func2()` is protected, so it cannot be preempted. The compiler always calls `func2()` that is defined in the shared library `libtest.so`. The visibility attribute of function `func3()` is hidden, so it is not exported in the shared library. The compiler issues a link error to indicate that the definition of `func3()` cannot be found. The same issue is with function `func4()` whose visibility attribute is internal.

Rules of visibility attributes

Rules of determining the visibility attributes

The visibility attribute of an entity is determined by the following rules:

1. If the entity has an explicitly specified visibility attribute, the specified visibility attribute takes effect.
2. Otherwise, if the entity has a pair of enclosing pragma directives, the visibility attribute that is specified by the pragma directives takes effect.
3. Otherwise, the setting of the **-qvisibility** option takes effect.

Note: On the AIX platform, if the setting of the **-qvisibility** option takes effect and is specified to **unspecified** (the default value), the entity does not have a visibility attribute.

Rules and restrictions of using the visibility attributes

When you specify visibility attributes for entities, consider the following rules and restrictions:

- You can specify visibility attributes only for entities that have external linkage. The compiler issues a warning message when you set the visibility attribute for entities with other linkages, and the specified visibility attribute is ignored. See Example 4.
- You cannot specify different visibility attributes in the same declaration or definition of an entity; otherwise, the compiler issues an error message. See Example 5.
- If an entity has more than one declaration that is specified with different visibility attributes, the visibility attribute of the entity is the first visibility attribute that the compiler processes. See Example 6.
- You cannot specify visibility attributes in the `typedef` statements. See Example 7.

Example 4

In this example, because `m` and `i` have internal linkage and `j` has no linkage, the compiler ignores the visibility attributes of variables `m`, `i`, and `j`.

```
static int m __attribute__((visibility("protected")));
int n __attribute__((visibility("protected")));

int main(){
   int i __attribute__((visibility("protected")));
   static int j __attribute__((visibility("protected)));
}
```

Example 5

In this example, the compiler issues an error message to indicate that you cannot specify two different visibility attributes at the same time in the definition of variable `m`.

```
//error
int m __attribute__((visibility("hidden"))) __attribute__((visibility("protected")));
```

Example 6

In this example, the first declaration of function `fun()` that the compiler processes is `extern void fun() __attribute__((visibility("hidden")))`, so the visibility attribute of `fun()` is hidden.

```
extern void fun() __attribute__((visibility("hidden")));
extern void fun() __attribute__((visibility("protected")));

int main(){
   fun();
}
```

Example 7

In this example, the visibility attribute of variable `vis_v_ti` is default, which is not affected by the setting in the `typedef` statement.

```
//The -qvisibility=default option is specified.
typedef int __attribute__((visibility("protected"))) INT;
INT vis_v_ti = 1;
```

Specifying visibility attributes using the `-qvisibility` option

You can use the **`-qvisibility`** option to globally set visibility attributes for external linkage entities in your program. The entities have the visibility attribute that is specified by the **`-qvisibility`** option if they do not get visibility attributes from pragma directives, explicitly specified attributes, or propagation rules.

Specifying visibility attributes using pragma preprocessor directives

You can selectively set visibility attributes for entities by using pairs of the `#pragma GCC visibility push` and `#pragma GCC visibility pop` preprocessor directives throughout your source program.

The compiler supports nested visibility pragma preprocessor directives. If entities are included in several pairs of the nested `#pragma GCC visibility push` and `#pragma GCC visibility pop` directives, the nearest pair of directives takes effect. See Example 1.

You must not specify the visibility pragma directives for header files. Otherwise, your program might exhibit undefined behaviors. See Example 2.

Examples

Example 1

In this example, the function and variables have the visibility attributes that are specified by their nearest pairs of pragma preprocessor directives.

```
#pragma GCC visibility push(default)
namespace ns
{
         void vis_f_fun() {}       //default
#  pragma GCC visibility push(internal)
         int vis_v_i;              //internal
#    pragma GCC visibility push(protected)
         int vis_v_j;              //protected
#      pragma GCC visibility push(hidden)
         int vis_v_k;              //hidden
#      pragma GCC visibility pop
#    pragma GCC visibility pop
#  pragma GCC visibility pop
}
#pragma GCC visibility pop
```

Example 2

In this example, the compiler issues a link error message to indicate that the definition of the `printf()` library function cannot be found.

```
#pragma GCC visibility push(hidden)
#include <stdio.h>
#pragma GCC visibility pop

int main(){
   printf("hello world!");
   return 0;
}
```

Chapter 10. Using the high performance libraries

IBM XL C for AIX, V13.1.2 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 that provide matrix/vector multiplication functions tuned for PowerPC architectures. The BLAS functions are described in "Using the Basic Linear Algebra Subprograms – BLAS" on page 108.

Using the Mathematical Acceleration Subsystem libraries (MASS)

XL C is shipped with a set of Mathematical Acceleration Subsystem (MASS) libraries for high-performance mathematical computing.

The MASS libraries consist of a library of scalar C functions described in "Using the scalar library" on page 96, a set of vector libraries tuned for specific architectures described in "Using the vector libraries" on page 98, and a set of SIMD libraries tuned for specific architectures described in "Using the SIMD libraries" on page 103. 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 accuracy and exception handling might not be identical in MASS functions and system library functions.

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

When you compile programs with any of the following sets of options:

- **-qhot -qignerrno -qnostrict**
- **-qhot -qignerrno -qstrict=nolibrary**
- **-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`, `vcosisin`, `vscosisin`, `vqdrt`, `vsqdrt`, `vrqdrt`, `vsrqdrt`, `vpopcnt4`, `vpopcnt8`, `vexp2`, `vexp2m1`, `vsexp2`, `vsexp2m1`, `vlog2`, `vlog21p`, `vslog2`, and `vslog21p`). If it cannot vectorize, it automatically tries to call the equivalent MASS scalar functions. For automatic vectorization or scalarization, the compiler uses versions of the MASS functions contained in the XLOPT library `libxlopt.a`.

In addition to any of the preceding sets of options, when the **-qipa** option is in effect, if the compiler cannot vectorize, it tries to inline the MASS scalar functions before deciding to call them.

"Compiling and linking a program with MASS" on page 107 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 website, 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`.

If you want to explicitly call the MASS scalar functions, you can take the following steps:

1. Provide the prototypes for the functions by including `math.h` and `mass.h` in your source files.
2. Link the MASS scalar library with your application. For instructions, see "Compiling and linking a program with MASS" on page 107.

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 25.

Table 25. 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);

Table 25. MASS scalar functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
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);
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 ones in the `libm.a` library, and they might handle edge cases differently (`sqrt(Inf)`, for example).
- For accuracy comparisons with `libm.a`, see Product documentation (manuals) in the Product support content section of the Mathematical Acceleration Subsystem website.
-

Related external information

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

Using the vector libraries

If you want to explicitly call any of the MASS vector functions, you can do so by including `massv.h` in your source files and linking your application with the appropriate vector library. Information about linking is provided in "Compiling and linking a program with MASS" on page 107.

libmassv.a
: The generic vector library that runs on any supported POWER® processor. Unless your application requires this portability, use the appropriate architecture-specific library below for maximum performance.

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.

libmassvp7.a
: Contains functions that have been tuned for the POWER7 architecture.

libmassvp8.a
: Contains functions that have been tuned for the POWER8 architecture.

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 Table 26 on page 99. The integer functions contained in the vector libraries are summarized in Table 27 on page 102. Note that in C applications, only call by reference is supported, even for scalar arguments.

With the exception of a few functions (described in the following paragraph), 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 an example, the following code:

```
#include <massv.h>

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

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

The functions `vdiv`, `vsincos`, `vpow`, and `vatan2` (and their single-precision versions, `vsdiv`, `vssincos`, `vspow`, and `vsatan2`) take four arguments. The functions `vdiv`, `vpow`, and `vatan2` take the arguments (*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 arguments (*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 26), the output vector holds y[0][i] = cos(x[i]) and y[1][i] = sin(x[i]), where i=0,..,*n-1.

Table 26. 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);

Table 26. MASS floating-point vector functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
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);
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);	
verf	vserf	Sets `y[i]` to the error function of `x[i]`, for i=0,..,*n-1	void verf (double y[], double x[], int *n)	void vserf (float y[], float x[], int *n)
verfc	vserfc	Sets `y[i]` to the complimentary error function of `x[i]`, for i=0,..,*n-1	void verfc (double y[], double x[], int *n)	void vserfc (float y[], float 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);
vexp2	vsexp2	Sets `y[i]` to 2 raised to the power of `x[i]`, for i=1,..,*n-1	void vexp2 (double y[], double x[], int *n);	void vsexp2 (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);
vexp2m1	vsexp2m1	Sets `y[i]` to (2 raised to the power of `x[i]`) - 1, for i=1,..,*n-1	void vexp2m1 (double y[], double x[], int *n);	void vsexp2m1 (float y[], float x[], int *n);
vhypot	vshypot	Sets `z[i]` to the square root of the sum of the squares of `x[i]` and `y[i]`, for i=0,..,*n-1	void vhypot (double z[], double x[], double y[], int *n)	void vshypot (float z[], float x[], float y[], 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);
vlog2	vslog2	Sets `y[i]` to the base-2 logarithm of `x[i]`, for i=1,..,*n-1	void vlog2 (double y[], double x[], int *n);	void vslog2 (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);

Table 26. MASS floating-point vector functions (continued)

Double-precision function	**Single-precision function**	**Description**	**Double-precision function prototype**	**Single-precision function prototype**
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);
vlog21p	vslog21p	Sets `y[i]` to the base-2 logarithm of (`x[i]`+1), for i=1,..,*n-1	void vlog21p (double y[], double x[], int *n);	void vslog21p (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);
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 does not compile on older versions of the operating system. To get an alternate prototype for these functions, compile with **`-D__nocomplex`**. This defines 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 27. 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 might 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 (x, y1, y2, &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.

Alignment of input and output vectors

To get the best performance from the POWER7 and POWER8 vector libraries, align the input and output vectors on 8-byte (or better, 16-byte) boundaries.

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 varies 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.

All the functions in `libmassvp7.a` and `libmassvp8.a` are consistent.

The following functions are consistent in all versions of the library in which they appear.

double-precision functions
: `vacos`, `vacosh`, `vasin`, `vasinh`, `vatan2`, `vatanh`, `vcbrt`, `vcos`, `vcosh`, `vcosisin`, `vdint`, `vdnint`, `vexp2`, `vexpm1`, `vexp2m1`, `vlog`, `vlog2`, `vlog10`, `vlog1p`, `vlog21p`, `vpow`, `vqdrt`, `vrcbrt`, `vrqdrt`, `vsin`, `vsincos`, `vsinh`, `vtan`, `vtanh`

single-precision functions
: `vsacos`, `vsacosh`, `vsasin`, `vsasinh`, `vsatan2`, `vsatanh`, `vscbrt`, `vscos`, `vscosh`, `vscosisin`, `vsexp`, `vsexp2`, `vsexpm1`, `vsexp2m1`, `vslog`, `vslog2`, `vslog10`, `vslog1p`, `vslog21p`, `vspow`, `vsqdrt`, `vsrcbrt`, `vsrqdrt`, `vssin`, `vssincos`, `vssinh`, `vssqrt`, `vstan`, `vstanh`

The following functions are consistent in `libmassvp3.a`, `libmassvp4.a`, `libmassvp5.a`, and `libmassvp6.a`:

`vsqrt` and `vrsqrt`.

The following functions are consistent in `libmassvp4.a`, `libmassvp5.a`, and `libmassvp6.a`:

`vrec`, `vsrec`, `vdiv`, `vsdiv`, and `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 website*. If consistency is not required, there might 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 website*.

Related information in the *XL C Compiler Reference*

-D

Related external information

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

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

Using the SIMD libraries

The MASS SIMD library libmass_simdp7.a or libmass_simdp8.a contains a set of frequently used math intrinsic functions that provide improved performance over the corresponding standard system library functions. If you want to use the MASS SIMD functions, you can do so as follows:

1. Provide the prototypes for the functions by including mass_simd.h in your source files.
2. Link the MASS SIMD library libmass_simdp7.a or libmass_simdp8.a with your application. For instructions, see "Compiling and linking a program with MASS" on page 107.

The single-precision MASS SIMD functions accept single-precision arguments and return single-precision results. Likewise, the double-precision MASS SIMD functions accept double-precision arguments and return double-precision results. They are summarized in Table 28.

Table 28. MASS SIMD functions

Double-precision function	**Single-precision function**	**Description**	**Double-precision function prototype**	**Single-precision function prototype**
acosd2	acosf4	Computes the arc cosine of each element of vx.	vector double acosd2 (vector double vx);	vector float acosf4 (vector float vx);
acoshd2	acoshf4	Computes the arc hyperbolic cosine of each element of vx.	vector double acoshd2 (vector double vx);	vector float acoshf4 (vector float vx);
asind2	asinf4	Computes the arc sine of each element of vx.	vector double asind2 (vector double vx);	vector float asinf4 (vector float vx);
asinhd2	asinhf4	Computes the arc hyperbolic sine of each element of vx.	vector double asinhd2 (vector double vx);	vector float asinhf4 (vector float vx);
atand2	atanf4	Computes the arc tangent of each element of vx.	vector double atand2 (vector double vx);	vector float atanf4 (vector float vx);
atan2d2	atan2f4	Computes the arc tangent of each element of vx/vy.	vector double atan2d2 (vector double vx, vector double vy);	vector float atan2f4 (vector float vx, vector float vy);
atanhd2	atanhf4	Computes the arc hyperbolic tangent of each element of vx.	vector double atanhd2 (vector double vx);	vector float atanhf4 (vector float vx);
cbrtd2	cbrtf4	Computes the cube root of each element of vx.	vector double cbrtd2 (vector double vx);	vector float cbrtf4 (vector float vx);
cosd2	cosf4	Computes the cosine of each element of vx.	vector double cosd2 (vector double vx);	vector float cosf4 (vector float vx);
coshd2	coshf4	Computes the hyperbolic cosine of each element of vx.	vector double coshd2 (vector double vx);	vector float coshf4 (vector float vx);

Table 28. MASS SIMD functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
cosisind2	cosisinf4	Computes the cosine and sine of each element of `x`, and stores the results in `y` and `z` as follows: `cosisind2 (x,y,z)` sets y and z to {`cos(x1), sin(x1)`} and {`cos(x2), sin(x2)`} where `x={x1,x2}`. `cosisinf4 (x,y,z)` sets y and z to {`cos(x1), sin(x1), cos(x2), sin(x2)`} and {`cos(x3), sin(x3), cos(x4), sin(x4)`} where `x={x1,x2,x3,x4}`.	void cosisind2 (vector double x, vector double *y, vector double *z)	void cosisinf4 (vector float x, vector float *y, vector float *z)
divd2	divf4	Computes the quotient `vx/vy`.	vector double divd2 (vector double vx, vector double vy);	vector float divf4 (vector float vx, vector float vy);
erfcd2	erfcf4	Computes the complementary error function of each element of `vx`.	vector double erfcd2 (vector double vx);	vector float erfcf4 (vector float vx);
erfd2	erff4	Computes the error function of each element of `vx`.	vector double erfd2 (vector double vx);	vector float erff4 (vector float vx);
expd2	expf4	Computes the exponential function of each element of `vx`.	vector double expd2 (vector double vx);	vector float expf4 (vector float vx);
exp2d2	exp2f4	Computes 2 raised to the power of each element of `vx`.	vector double exp2d2 (vector double vx);	vector float exp2f4 (vector float vx);
expm1d2	expm1f4	Computes (the exponential function of each element of `vx`) - 1.	vector double expm1d2 (vector double vx);	vector float expm1f4 (vector float vx);
exp2m1d2	exp2m1f4	Computes (2 raised to the power of each element of `vx`) -1.	vector double exp2m1d2 (vector double vx);	vector float exp2m1f4 (vector float vx);
hypotd2	hypotf4	For each element of `vx` and the corresponding element of `vy`, computes `sqrt(x*x+y*y)`.	vector double hypotd2 (vector double vx, vector double vy);	vector float hypotf4 (vector float vx, vector float vy);
lgammad2	lgammaf4	Computes the natural logarithm of the absolute value of the Gamma function of each element of `vx` .	vector double lgammad2 (vector double vx);	vector float lgammaf4 (vector float vx);

Table 28. MASS SIMD functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
logd2	logf4	Computes the natural logarithm of each element of vx.	vector double logd2 (vector double vx);	vector float logf4 (vector float vx);
log2d2	log2f4	Computes the base-2 logarithm of each element of vx.	vector double log2d2 (vector double vx);	vector float log2f4 (vector float vx);
log10d2	log10f4	Computes the base-10 logarithm of each element of vx.	vector double log10d2 (vector double vx);	vector float log10f4 (vector float vx);
log1pd2	log1pf4	Computes the natural logarithm of each element of (vx +1).	vector double log1pd2 (vector double vx);	vector float log1pf4 (vector float vx);
log21pd2	log21pf4	Computes the base-2 logarithm of each element of (vx +1).	vector double log21pd2 (vector double vx);	vector float log21pf4 (vector float vx);
powd2	powf4	Computes each element of vx raised to the power of the corresponding element of vy.	vector double powd2 (vector double vx, vector double vy);	vector float powf4 (vector float vx, vector float vy);
qdrtd2	qdrtf4	Computes the quad root of each element of vx.	vector double qdrtd2 (vector double vx);	vector float qdrtf4 (vector float vx);
rcbrtd2	rcbrtf4	Computes the reciprocal of the cube root of each element of vx.	vector double rcbrtd2 (vector double vx);	vector float rcbrtf4 (vector float vx);
recipd2	recipf4	Computes the reciprocal of each element of vx.	vector double recipd2 (vector double vx);	vector float recipf4 (vector float vx);
rqdrtd2	rqdrtf4	Computes the reciprocal of the quad root of each element of vx.	vector double rqdrtd2 (vector double vx);	vector float rqdrtf4 (vector float vx);
rsqrtd2	rsqrtf4	Computes the reciprocal of the square root of each element of vx.	vector double rsqrtd2 (vector double vx);	vector float rsqrtf4 (vector float vx);
sincosd2	sincosf4	Computes the sine and cosine of each element of vx.	void sincosd2 (vector double vx, vector double *vs, vector double *vc);	void sincosf4 (vector float vx, vector float *vs, vector float *vc);
sind2	sinf4	Computes the sine of each element of vx.	vector double sind2 (vector double vx);	vector float sinf4 (vector float vx);
sinhd2	sinhf4	Computes the hyperbolic sine of each element of vx.	vector double sinhd2 (vector double vx);	vector float sinhf4 (vector float vx);
sqrtd2	sqrtf4	Computes the square root of each element of vx.	vector double sqrtd2 (vector double vx);	vector float sqrtf4 (vector float vx);

Table 28. MASS SIMD functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
tand2	tanf4	Computes the tangent of each element of vx.	vector double tand2 (vector double vx);	vector float tanf4 (vector float vx);
tanhd2	tanhf4	Computes the hyperbolic tangent of each element of vx.	vector double tanhd2 (vector double vx);	vector float tanhf4 (vector float vx);

Compiling and linking a program with MASS

To compile an application that calls the functions in the scalar, SIMD, or vector MASS libraries, specify **`mass`**, and/or one of **`mass_simdp7`**, **`mass_simdp8`**, and/or one of **`massv`**, **`massvp4`**, **`massvp5`**, **`massvp6`**, **`massvp7`**, **`massvp8`** on the **`-l`** linker option.

For example, if the MASS libraries are installed in the default directory, you can specify one of the following:

Link object file `progc` with scalar library libmass.a and vector library libmassvp8.a

```
xlc -qarch=pwr8 progc.c -o progc -lmass -lmassvp8
```

Link object file `progc` with SIMD library libmass_simdp8.a

```
xlc -qarch=pwr8 progc.c -o progc -lmass_simdp8
```

Using libmass.a with the math system library

If you want 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 that is a flat text file and contains the names of the wanted 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
   ```

Notes:

- The MASS `sincos` function is automatically linked if you export MASS `cosisin`.
- The MASS `cos` function is automatically linked if you export MASS `sin`.
- The 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 the XL C compiler 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 information and examples of usage of these functions, see *Engineering and Scientific Subroutine Library Guide and Reference*, available at the Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL web page.

"Linking the libxlopt library" on page 110 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 input matrix *a*, where:
  - `'N'` or `'n'` indicates that *a* is to be used in computation
  - `'T'` or `'t'` indicates that the transpose of *a* is to be used in 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 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 input matrix *a*, where:
- `'N'` or `'n'` indicates that *a* is to be used in computation
- `'T'` or `'t'` indicates that the transpose of *a* is to be used in computation

transb
: is a single character indicating the form of input matrix *b*, where:
- `'N'` or `'n'` indicates that *b* is to be used in computation
- `'T'` or `'t'` indicates that the transpose of *b* is to be used in 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 that you compile with the XL C compiler. 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 the following methods of implementing shared memory program parallelization:

- Automatic parallelization of countable program loops, which are defined in "Countable loops" on page 112. An overview of the compiler's automatic parallelization capabilities is provided in "Enabling automatic parallelization" on page 113.
- 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 113.
- 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 116.

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 about 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 about OpenMP environment variables, see *OpenMP environment variables for parallel processing* in the *XL C Compiler Reference*. For more information about both IBM SMP and OpenMP built-in functions, see *Built-in functions for parallel processing* in the *XL C Compiler Reference*.

For details about the OpenMP constructs, environment variables, and runtime routines, refer to the *OpenMP Application Program Interface Specification*, available at http://www.openmp.org.

Related information:

"Using shared-memory parallelism (SMP)" on page 54

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 these 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 parallelize all countable loops where possible in your program code. A loop is considered to be countable if it has any of the forms shown in "Countable loops" on page 112, 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 invocation mode (those that contain the **_r** suffix).

Using IBM SMP directives

Note: The pragma directive **#pragma ibm schedule** has been deprecated and might be removed in a future release. You can use the corresponding OpenMP directives or clauses to obtain the same behavior.

For detailed information about how to replace the deprecated pragma directives with corresponding OpenMP directives, refer to "Deprecated directives" in the *XL C Compiler Reference*.

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 112. 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 sequential_loop**), apply specific parallelization algorithms to a loop (**#pragma ibm schedule**), and synchronize access to shared variables using critical sections.

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, for example, the **parallel_loop** and **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

Data sharing attribute rules

The rules of data sharing attributes determine the attributes of variables that are referenced in `parallel` and `task` directives, and worksharing regions.

Data sharing attribute rules for variables referenced in a construct

The data sharing attributes of variables that are referenced in a construct can be classified into the following categories:

- Predetermined data sharing attributes
- Explicitly determined data sharing attributes
- Implicitly determined data sharing attributes

Specifying a variable in a `firstprivate`, `lastprivate`, or `reduction` clause of an enclosed construct initiates an implicit reference to the variable in the enclosing construct. Such implicit references also follow the data sharing attribute rules.

Some variables and objects have predetermined data sharing attributes as follows:

- Variables that are specified in `threadprivate` directives are threadprivate.
- Variables with automatic storage duration that are declared in a scope inside the construct are private.
- Objects with dynamic storage duration are shared.
- Static data members are shared.
- The loop iteration variables in the associated `for` loops of a `for` or `parallel for` construct are private.
- Variables with `const`-qualified types are shared if they have no mutable member.
- For variables with static storage duration, if they are declared in a scope inside the construct, they are shared.

Variables with predetermined data sharing attributes cannot be specified in data sharing attribute clauses. However, in the following situations, specifying a predetermined variable in a data sharing attribute clause is allowed and overrides the predetermined data sharing attributes of the variable.

- The loop iteration variables in the associated `for` loops of a `for` or `parallel for` construct can be specified in a `private` or `lastprivate` clause.
- For variables with `const`-qualified type, if they have no mutable member, they can be specified in a `firstprivate` clause.

Variables that meet the following conditions have explicitly determined data sharing attributes:

- The variables are referenced in a construct.
- The variables are specified in a data sharing attribute clause on the construct.

Variables that meet all the following conditions have implicitly determined data sharing attributes:

- The variables are referenced in a construct.
- The variables do not have predetermined data sharing attributes.
- The variables are not specified in a data sharing attribute clause on the construct.

For variables that have implicitly determined data sharing attributes, the rules are as follows:

- In a `parallel` or `task` construct, the data sharing attributes of the variables are determined by the `default` clause, if present.
- In a `parallel` construct, if no `default` clause is present, the variables are shared.
- For constructs other than `task`, if no `default` clause is present, the variables inherit their data sharing attributes from the enclosing context.
- In a `task` construct, if no `default` clause is present, variables that are determined to be shared in the enclosing context by all implicit tasks bound to the current team are shared.
- In a `task` construct, if no `default` clause is present, variables whose data sharing attributes are not determined by the rules above are firstprivate.

Data sharing attribute rules for variables referenced in a region but not in a construct

The data sharing attributes of variables that are referenced in a region, but not in a construct, are determined as follows:

- If variables with static storage duration are declared in called routines in the region, the variables are shared.
- Variables with `const`-qualified types are shared if they have no mutable member and are declared in called routines.
- File-scope or namespace-scope variables referenced in called routines in the region are shared unless they are specified in a `threadprivate` directive.
- Objects with dynamic storage duration are shared.
- Static data members are shared unless they are specified in a `threadprivate` directive.
- The formal arguments of called routines in the region that are passed by reference inherit the data sharing attributes of the associated actual arguments.
- Other variables declared in called routines in the region are private.

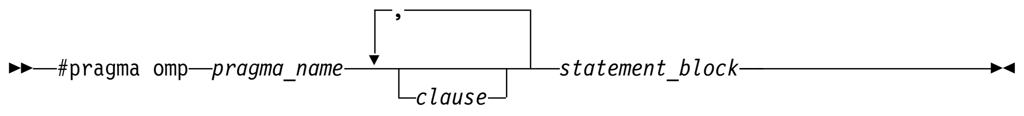
Using OpenMP directives

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.

The **`#pragma omp`** pragmas fall into these general categories:

1. The **`#pragma omp`** 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. The **`#pragma omp`** pragmas that let you define how work is distributed or shared across the threads in a parallel region (**`#pragma omp sections`**, **`#pragma omp for`**, **`#pragma omp single`**, **`#pragma omp task`**).
3. The **`#pragma omp`** pragmas that let you control synchronization among threads (**`#pragma omp atomic`**, **`#pragma omp master`**, **`#pragma omp barrier`**, **`#pragma omp critical`**, **`#pragma omp flush`**, **`#pragma omp ordered`**) .
4. The **`#pragma omp`** pragmas that let you define the scope of data visibility across parallel regions within the same thread (**`#pragma omp threadprivate`**).
5. The **`#pragma omp`** pragmas for synchronization (**`#pragma omp taskwait`**, **`#pragma omp barrier`**)

OpenMP directive syntax



Adding certain clauses to the **`#pragma omp`** pragmas can fine tune the behavior of the parallel or work-sharing regions. For example, a `num_threads` clause can be used to control a parallel region pragma.

The **`#pragma omp`** pragmas generally appear immediately before the section of code to which they apply. 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 parallel
{
  #pragma omp sections
  {
    #pragma omp section
       structured_block_1
           ...
    #pragma omp section
       structured_block_2
           ...
        ....
  }
}
```

For a pragma-by-pragma description of the OpenMP 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 regions. 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 that are declared in a parallel region 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 region 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 */
 ... }
```

Some OpenMP clauses 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. The private variable is initialized by the original value of the variable when the parallel construct is encountered.
lastprivate	The **lastprivate** clause provides a superset of the functionality provided by the **private** clause. The private variable is updated after the end of the parallel construct.
shared	The **shared** clause declares the variables in the list to be shared 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-sharing attribute of the variables appeared in the parallel construct.

For more information, see the OpenMP directive descriptions in "Pragma directives for parallel processing" in the *XL C Compiler Reference*. You can also refer to the *OpenMP Application Program Interface Language Specification*, which is available at http://www.openmp.org.

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:

```
►►—variable—=—variable—┬─+─────┬—expression———————————————————————►◄
                       ├─-─────┤
                       ├─*─────┤
                       ├─^─────┤
                       ├─|─────┤
                       └─&    ─┘
```

```
►►—variable——┬─+=────┬—expression——————————————————————————————►◄
             ├─-=────┤
             ├─*=────┤
             ├─^=────┤
             ├─|=────┤
             └─&=   ─┘
```

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.

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 compiler 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 130 describes the debug versions of the standard library functions for manipulating strings.

Notes:

- The memory debug library supports only extensions for the memory management functions that are described in this document.
- The compiler supports the memory allocation debug functions, but IBM has no plans to change or enhance these functions, and these functions will be removed in a future release. If you use these functions to debug memory problems in your programs, you can migrate to the AIX debug malloc tool to achieve equivalent functionality. For details of the AIX debug malloc tool, see http://publib16.boulder.ibm.com/pseries/index.htm.

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

- In your source code, prefix any of the default or user-defined-heap memory management functions with `_debug_`.
- If you do not want to make changes to the source code, 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 35

_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 is similar to:

```
1546-503 End of allocated object 0x2001D9F0 was overwritten at 0x2001DA54.
1546-514 The first eight bytes of the object (in hex) are: 6161616161616161.
1546-519 This object was (re)allocated at line 10 in _debug_calloc.c.
       _debug_ucalloc@AF53_46 + 6C
          _debug_ucalloc_init + 44
                _debug_calloc + 64
                         main + 28
1546-512 Heap state was valid at line 10 in _debug_calloc.c.
           _int_debug_umalloc + 54
       _debug_ucalloc@AF53_46 + 6C
          _debug_ucalloc_init + 44
                _debug_calloc + 64
                         main + 28
1546-511 Heap error detected at line 14 in _debug_calloc.c.
1546-522 Traceback:
           d0c017a4 = _debug_memset + 0x50
           100003ec = main + 0x78
```

_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` 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 is similar to:

```
1546-507 Free heap was overwritten at 0x2001DA10.
1546-512 Heap state was valid at line 14 in _debug_free.c.
                _debug_ufree + CC
                 _debug_free + 10
                        main + 8C
1546-511 Heap error detected at line 15 in _debug_free.c.
1546-522 Traceback:
           d0c017a4 = _debug_memset + 0x50
           10000424 = main + 0xB0
```

_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;
}
```

The output is similar to:

```
1546-510 Header information of object 0x200360E0 was overwritten at 0x200360DC.
1546-514 The first eight bytes of the object (in hex) are: 4300000000000000.
1546-519 This object was (re)allocated at line 14 in _debug_heapmin.c.
              _debug_urealloc + 128
               _debug_realloc + 18
                         main + 6C
1546-512 Heap state was valid at line 15 in _debug_heapmin.c.
              _debug_uheapmin + 50
               _debug_heapmin + 54
                         main + 80
1546-511 Heap error detected at line 18 in _debug_heapmin.c.
1546-522 Traceback:
           d119522c = _debug_uheapmin + 0x58
           d0c010d0 = _debug_heapmin + 0x5C
           10000418 = main + 0xA4
```

_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;
}
```

The output is similar to:

```
1546-510 Header information of object 0x2001D9C0 was overwritten at 0x2001D9BC.
1546-514 The first eight bytes of the object (in hex) are: 4300000000000000.
1546-519 This object was (re)allocated at line 9 in _debug_malloc.c.
               _debug_umalloc + D0
          _debug_umalloc_init + 3C
                _debug_malloc + 5C
                         main + 24
1546-512 Heap state was valid at line 9 in _debug_malloc.c.
           _int_debug_umalloc + 54
               _debug_umalloc + D0
          _debug_umalloc_init + 3C
                _debug_malloc + 5C
                         main + 24
1546-511 Heap error detected at line 14 in _debug_malloc.c.
1546-522 Traceback:
           d1195358 = _debug_umalloc + 0x84
           d0c01258 = _debug_malloc + 0x64
           100003ec = main + 0x78
```

_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 is similar to :

```
1546-503 End of allocated object 0x2001DA00 was overwritten at 0x2001DA64.
1546-514 The first eight bytes of the object (in hex) are: 7878787878787878.
1546-519 This object was (re)allocated at line 15 in _debug_ucalloc.c.
       _debug_ucalloc@AF53_46 + 6C
                       main + 38
```

```
1546-512 Heap state was valid at line 15 in _debug_ucalloc.c.
           _int_debug_umalloc + 54
       _debug_ucalloc@AF53_46 + 6C
                        main + 38
1546-511 Heap error detected at line 20 in _debug_ucalloc.c.
1546-522 Traceback:
          d11954e8 = _debug_ufree + 0xD4
          d0c0112c = _debug_free + 0x18
          10000410 = main + 0x9C
```

_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 is similar to :

```
1546-503 End of allocated object 0x2001DA00 was overwritten at 0x2001DA64.
1546-514 The first eight bytes of the object (in hex) are: 7878787878787878.
1546-519 This object was (re)allocated at line 15 in _debug_umalloc.c.
               _debug_umalloc + D0
                         main + 34
1546-512 Heap state was valid at line 15 in _debug_umalloc.c.
           _int_debug_umalloc + 54
               _debug_umalloc + D0
                         main + 34
1546-511 Heap error detected at line 20 in _debug_umalloc.c.
1546-522 Traceback:
           d11954e8 = _debug_ufree + 0xD4
           d0c0112c = _debug_free + 0x18
           1000040c = main + 0x98
```

_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 is similar to:

```
1546-503 End of allocated object 0x2001D9F0 was overwritten at 0x2001DA54.
1546-514 The first eight bytes of the object (in hex) are: 6161616161616161.
1546-519 This object was (re)allocated at line 10 in _debug_realloc.c.
               _debug_umalloc + D0
          _debug_umalloc_init + 3C
                _debug_malloc + 5C
                         main + 28
1546-512 Heap state was valid at line 10 in _debug_realloc.c.
           _int_debug_umalloc + 54
               _debug_umalloc + D0
          _debug_umalloc_init + 3C
                _debug_malloc + 5C
                         main + 28
1546-511 Heap error detected at line 14 in _debug_realloc.c.
1546-522 Traceback:
           d0c017a4 = _debug_memset + 0x50
           100003ec = main + 0x78
```

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 reports 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 is similar to:

```
1546-503 End of allocated object 0x2001D9B0 was overwritten at 0x2001D9BA.
1546-514 The first eight bytes of the object (in hex) are: 5468697320697320.
1546-519 This object was (re)allocated at line 12 in _debug_memcpy.c.
               _debug_umalloc + D0
          _debug_umalloc_init + 3C
                _debug_malloc + 5C
                         main + 24
1546-512 Heap state was valid at line 12 in _debug_memcpy.c.
           _int_debug_umalloc + 54
               _debug_umalloc + D0
          _debug_umalloc_init + 3C
                _debug_malloc + 5C
                         main + 24
1546-511 Heap error detected at line 13 in _debug_memcpy.c.
1546-522 Traceback:
           d0c018a4 = _debug_memcpy + 0x50
           100003bc = main + 0x48
```

_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` reports 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 is similar to:

```
Buffer contents: AAAAAAAAAA
1546-503 End of allocated object 0x2001D9A0 was overwritten at 0x2001D9B5.
1546-514 The first eight bytes of the object (in hex) are: 4141414141414141.
1546-519 This object was (re)allocated at line 13 in _debug_memset.c.
         _debug_ucalloc@AF53_46 + 6C
            _debug_ucalloc_init + 44
                  _debug_calloc + 64
                           main + 28
1546-512 Heap state was valid at line 15 in _debug_memset.c.
                   _chk_if_heap + E8
                  _debug_memset + 48
                           main + 44
```

```
1546-511 Heap error detected at line 17 in _debug_memset.c.
1546-522 Traceback:
          d0c017a4 = _debug_memset + 0x50
          100003f4 = main + 0x80
```

_debug_strcat — Concatenate strings

Format

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

Purpose

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` reports the file name *file* and line number *file* in a message.

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.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");

   ptr = strcat(buffer1, " program");
   printf("buffer1 = %s\n", buffer1);
   return 0;
}
```

The output is similar to:

```
1546-503 End of allocated object 0x2001D9D0 was overwritten at 0x2001D9DA.
1546-514 The first eight bytes of the object (in hex) are: 636F6D7075746572.
1546-519 This object was (re)allocated at line 13 in _debug_strcat.c.
              _debug_umalloc + D0
         _debug_umalloc_init + 3C
               _debug_malloc + 5C
                        main + 24
1546-512 Heap state was valid at line 14 in _debug_strcat.c.
                _chk_if_heap + E8
                _debug_strcpy + 50
                         main + 3C
1546-511 Heap error detected at line 16 in _debug_strcat.c.
1546-522 Traceback:
           d0c01630 = _debug_strcat + 0x9C
           100003d0 = main + 0x5C
```

_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` reports 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 is similar to:

```
destination is originally = 'abcdefg'
1546-503 End of allocated object 0x2001D9F0 was overwritten at 0x2001D9FA.
1546-514 The first eight bytes of the object (in hex) are: 3132333435363738.
1546-519 This object was (re)allocated at line 14 in _debug_strcpy.c.
               _debug_umalloc + D0
          _debug_umalloc_init + 3C
                _debug_malloc + 5C
                         main + 2C
```

```
1546-512 Heap state was valid at line 15 in _debug_strcpy.c.
                _chk_if_heap + E8
               _debug_strcpy + 50
                        main + 44
1546-511 Heap error detected at line 18 in _debug_strcpy.c.
1546-522 Traceback:
           d0c0170c = _debug_strcpy + 0x58
           100003e8 = main + 0x74
```

_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` reports 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 is similar to:

```
1546-503 End of allocated object 0x2001D9E0 was overwritten at 0x2001D9EA.
1546-514 The first eight bytes of the object (in hex) are: 636F6D7075746572.
1546-519 This object was (re)allocated at line 13 in _debug_strncat.c.
```

```
               _debug_umalloc + D0
          _debug_umalloc_init + 3C
               _debug_malloc + 5C
                        main + 24
1546-512 Heap state was valid at line 14 in _debug_strncat.c.
                _chk_if_heap + E8
               _debug_strcpy + 50
                        main + 3C
1546-511 Heap error detected at line 18 in _debug_strncat.c.
1546-522 Traceback:
           d0c01518 = _debug_strncat + 0xC4
           100003d4 = main + 0x60
```

_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 reports 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_strncpy.c  */
#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 is similar to:

```
destination is originally = 'abcdefg'
1546-503 End of allocated object 0x2001DA10 was overwritten at 0x2001DA1A.
1546-514 The first eight bytes of the object (in hex) are: 3132333435363738.
1546-519 This object was (re)allocated at line 15 in _debug_strncpy.c.
               _debug_umalloc + D0
          _debug_umalloc_init + 3C
                _debug_malloc + 5C
                         main + 34
1546-512 Heap state was valid at line 16 in _debug_strncpy.c.
                 _chk_if_heap + E8
                _debug_strcpy + 50
                         main + 4C
1546-511 Heap error detected at line 19 in _debug_strncpy.c.
1546-522 Traceback:
           d0c0137c = _debug_strncpy + 0x48
           100003f4 = main + 0x80
```

Notices

Programming interfaces: Intended programming interfaces allow the customer to write programs to obtain the services of IBM XL C for AIX.

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

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

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
U.S.A.

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

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

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

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those

Trademarks

Index

Product Number: 5765-J06; 5725-C71

Printed in USA

SC27-4241-01