IBM XL C/C++ Advanced Edition V8.0 for Linux

Programming Guide

SC09-8014-00

IBM XL C/C++ Advanced Edition V8.0 for Linux

Programming Guide

SC09-8014-00

Note!
Before using this information and the product it supports, read the information in "Notices" on page 71.

First Edition (November, 2005)

This edition applies to version 8.0 of IBM XL C/C++ Advanced Edition V8.0 for Linux (product number 5724-M16) and to all subsequent releases and modifications until otherwise indicated in new editions.

IBM welcomes your comments. You can send them to compinfo@ca.ibm.com. Be sure to include your e-mail address if you want a reply. Include the title and order number of this book, and the page number or topic related to your comment.

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

Contents

About this document

This guide discusses advanced topics related to the use of the IBM® IBM XL C/C++ Advanced Edition V8.0 for Linux® compiler, with a particular focus on program portability and optimization. The guide provides both reference information and practical tips for getting the most out of the compiler's capabilities, through recommended programming practices and compilation procedures.

Who should read this document

This document is addressed to programmers building complex applications, who already have experience compiling with XL C/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 presenting the topics, by concentrating on a specific programming or compilation problem in each section. Each topic contains extensive cross-references to the relevant sections of the reference guides in the IBM XL C/C++ Advanced Edition V8.0 for Linux documentation set, which provide detailed descriptions of compiler options and pragmas, and specific language extensions.

How this document is organized

This guide includes these topics:

- Chapter 1, "Using 32-bit and 64-bit modes," on page 1 discusses common problems that arise when porting existing 32-bit applications to 64-bit mode, and provides recommendations for avoiding these problems.
- Chapter 2, "Using XL C/C++ with Fortran," on page 5 discussions considerations for calling Fortran code from XL C/C++ programs.
- Chapter 3, "Aligning data," on page 9 discusses the different compiler options available for controlling the alignment of data in aggregates, such as structures and classes, on all platforms.
- Chapter 4, "Handling floating point operations," on page 17 discusses options available for controlling the way floating-point operations are handled by the compiler.
- Chapter 5, "Using C++ templates," on page 21 discusses the different options for compiling programs that include C++ templates.
- Chapter 6, "Constructing a library," on page 27 discusses how to compile and link static and shared libraries, and how to specify the initialization order of static objects in C++ programs.
- Chapter 7, "Optimizing your applications," on page 35 discusses the various options provided by the compiler for optimizing your programs, and provides recommendations for use of the different options.

- Chapter 8, “Coding your application to improve performance,” on page 49 discusses recommended programming practices and coding techniques for enhancing program performance and compatibility with the compiler’s optimization capabilities.
- Chapter 9, “Using the high performance libraries,” on page 55 discusses two libraries that are shipped with XL C/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 10, “Parallelizing your programs,” on page 65 provides an overview of the different options offered by the IBM XL C/C++ Advanced Edition V8.0 for Linux for creating multi-threaded programs, including OpenMP language constructs.

Conventions and terminology used in this document

The following sections discuss the conventions used in this document:

Typographical conventions

The following table explains the typographical conventions used in this document.

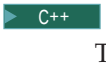
Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Commands, executable names, compiler options and pragma directives.	Use the **-qmkshrobj** compiler option to create a shared object from the generated object files.
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.
`monospace`	Programming keywords and library functions, compiler built-in functions, file and directory names, examples of program code, command strings, or user-defined names.	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.

Icons

All features described in this document apply to both C and C++ languages. Where a feature is exclusive to one language, or where functionality differs between languages, the following icons are used:

The text describes a feature that is supported in the C language only; or describes behavior that is specific to the C language.



The text describes a feature that is supported in the C++ language only; or describes behavior that is specific to the C++ language.

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

AIX

The text describes the functionality supported on the AIX® platform.

Linux

The text describes the functionality supported on the Linux® platform.

How to read syntax diagrams

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ▶▶— symbol indicates the beginning of a command, directive, or statement.

  The —▶ symbol indicates that the command, directive, or statement syntax is continued on the next line.

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

  The —▶◀ symbol indicates the end of a command, directive, or statement.

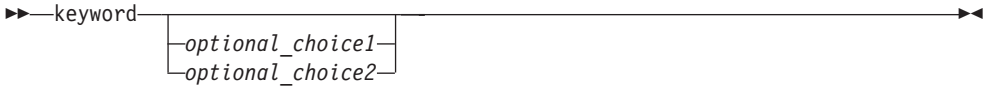
  Diagrams of syntactical units other than complete commands, directives, or statements start with the ▶— symbol and end with the —▶ symbol.
- Required items appear on the horizontal line (the main path).

- Optional items are shown below the main path.

- 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.

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



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

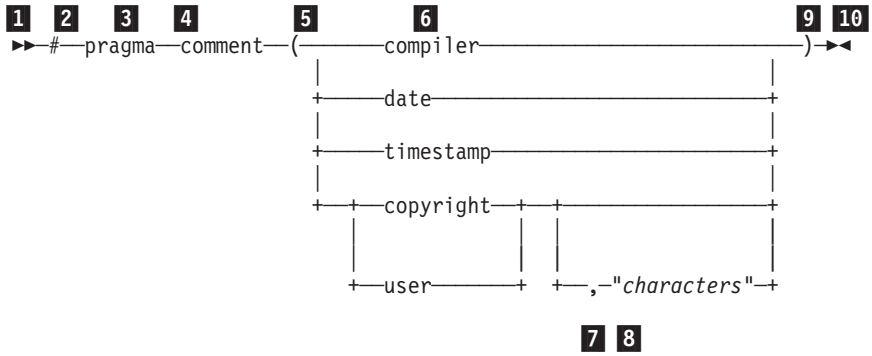
- An arrow returning to the left above the main line indicates an item that can be repeated.

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

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

  Variables are shown in italicized lowercase letters (for example, *identifier*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

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



1 This is the start of the syntax diagram.

2 The symbol # must appear first.

3 The keyword `pragma` must appear following the # symbol.

4 The name of the pragma `comment` must appear following the keyword `pragma`.

5 An opening parenthesis must be present.

6 The comment type must be entered only as one of the types indicated: `compiler`, `date`, `timestamp`, `copyright`, or `user`.

7 A comma must appear between the comment type `copyright` or `user`, and an optional character string.

8 A character string must follow the comma. The character string must be enclosed in double quotation marks.

9 A closing parenthesis is required.

10 This is the end of the syntax diagram.

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

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

Examples

The examples in this document, 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.

Related information

IBM XL C/C++ publications

XL XL C/C++ provides product documentation in the following formats:

- Readme files

  Readme files contain late-breaking information, including changes and corrections to the product documentation. Readme files are located by default in the /opt/ibmcmp/vacpp/8.0/ directory and in the root directory of the installation CD.

- Installable man pages

  Man pages are provided for the compiler invocations and all command-line utilities provided with the product. Instructions for installing and accessing the man pages are provided in the *IBM XL C/C++ Advanced Edition V8.0 for Linux Installation Guide*.

- Information center

  The information center of searchable HTML files can be launched on a network and accessed remotely or locally. Instructions for installing and accessing the information center are provided in the *IBM XL C/C++ Advanced Edition V8.0 for Linux Installation Guide*. The information center is also viewable on the Web at:

  http://publib.boulder.ibm.com/infocenter/lnxpcomp/index.jsp.

- PDF documents

  PDF documents are located by default in the /opt/ibmcmp/vacpp/8.0/doc/*language*/pdf/ directory, and are also available on the Web at:

  www.ibm.com/software/awdtools/xlcpp/library.

  In addition to this document, the following files comprise the full set of XL C/C++ product manuals:

Table 2. XL C/C++ PDF files

Document title	PDF file name	Description
IBM XL C/C++ Advanced Edition V8.0 for Linux Installation Guide, GC09-8017-00	install.pdf	Contains information for installing XL C/C++ and configuring your environment for basic compilation and program execution.
IBM XL C/C++ Advanced Edition V8.0 for Linux Getting Started Guide, SC09-8015-00	getstart.pdf	Contains an introduction to the XL C/C++ product, with information on setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
IBM XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference, SC09-8013-00	compiler.pdf	Contains information about the various compiler options, pragmas, macros, environment variables, and built-in functions, including those used for parallel processing.

Table 2. XL C/C++ PDF files (continued)

Document title	PDF file name	Description
IBM XL C/C++ Advanced Edition V8.0 for Linux Language Reference, SC09-8016-00	language.pdf	Contains information about the C and C++ programming languages, as supported by IBM, including language extensions for portability and conformance to non-proprietary standards.

These PDF files are viewable and printable from Adobe Reader. If you do not have the Adobe Reader installed, you can download it from www.adobe.com.

Additional documentation

More documentation related to XL C/C++, including redbooks, whitepapers, tutorials, and other articles, is available on the Web at:

www.ibm.com/software/awdtools/xlcpp/library

Related publications

You might want to consult the following publications, which are also referenced throughout this document:

- *OpenMP Application Program Interface Version 2.5*, available at www.openmp.org
- *ESSL for AIX V4.2 ESSL for Linux on POWER V4.2 Guide and Reference*, SA22-7904-02

Technical support

Additional technical support is available from the XL C/C++ Support page. This page provides a portal with search capabilities to a large selection of technical support FAQs and other support documents. You can find the XL C/C++ Support page on the Web at:

www.ibm.com/software/awdtools/xlcpp/support

If you cannot find what you need, you can e-mail:

compinfo@ca.ibm.com

For the latest information about XL C/C++, visit the product information site at:

www.ibm.com/software/awdtools/xlcpp

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 document or any other XL C/C++ documentation, send your comments by e-mail to:

compinfo@ca.ibm.com

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

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

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

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

Table 3. Size and alignment of data types in 32-bit and 64-bit modes

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

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

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

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

For suggestions on improving performance in 64-bit mode, see "Optimize operations in 64-bit mode" on page 53.

Related information

- **-q32**/**-q64** and **-qwarn64** in *XL C/C++ Compiler Reference*

Assigning long values

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

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

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

Implications of these differences are:

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

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

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

Assigning constant values to long variables

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

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

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

Table 5. Unexpected boundary results of constants assigned to long types

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

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

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

Bit-shifting long values

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

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

Table 6. Results of bit-shifting long values

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

Assigning pointers

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

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

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

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

Without a function prototype for `calloc`, when the same code is compiled in 64-bit mode, the compiler assumes the function returns an `int`, so `a` is silently truncated, and then sign-extended. Type casting the result will not prevent the truncation, as the address of the memory allocated by `calloc` was already truncated during the return. In this example, the correct solution would be to include the header file, `stdlib.h`, which contains the prototype for `calloc`.

To avoid these types of problems:

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

Aligning aggregate data

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

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

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

Calling Fortran code

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

Table 7. Equivalent C/C++ and Fortran data types

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

Related information

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

Chapter 2. Using XL C/C++ with Fortran

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

- "Identifiers"
- "Corresponding data types"
- "Character and aggregate data" on page 6
- "Function calls and parameter passing" on page 7
- "Pointers to functions" on page 7
-

"Sample program: C/C++ calling Fortran" on page 7 provides an example of a C program which calls a Fortran subroutine.

Related information

- "Calling Fortran code" on page 4

Identifiers

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

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

Corresponding data types

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

Table 8. Correspondence of data types among C, C++ and Fortran

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

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

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

C and C++ data types	Fortran data types
signed long long int	INTEGER*8
unsigned long long int	LOGICAL*8
float	REAL REAL*4
double	REAL*8 DOUBLE PRECISION
long double	REAL*8 DOUBLE PRECISION
float _Complex	COMPLEX*8 or COMPLEX(4)
double _Complex	COMPLEX*16 or COMPLEX(8)
long double _Complex	COMPLEX*16 or COMPLEX(8)
structure	—
enumeration	INTEGER*4
char[n]	CHARACTER*n
array pointer to type, or type []	Dimensioned variable (transposed)
pointer to function	Functional parameter
structure (with **-qalign=packed**)	Sequence derived type

Related information

- **-qlongdouble** and **-qldbl128**) in *XL C/C++ Compiler Reference*

Character and aggregate data

Most numeric data types have counterparts across C/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.
- C stores array elements in row-major order (array elements in the same row occupy adjacent memory locations). Fortran stores array elements in ascending storage units in column-major order (array elements in the same column occupy adjacent memory locations). Table 9 shows how a two-dimensional array declared by A[3][2] in C and by A(3,2) in Fortran, is stored:

Table 9. Storage of a two-dimensional array

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

Table 9. Storage of a two-dimensional array (continued)

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

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

Function calls and parameter passing

Functions must be prototyped identically in both C/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 or C++ functions that call a program written in Fortran, all arguments must be pointers, or scalars with the address operator.

Pointers to functions

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

Sample program: C/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/C++ and Fortran subroutines with different data types as arguments.

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

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

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

x = 3;
y = 3;
```

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

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

The Fortran subroutine is:

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

C Compile separately, then link to C/C++ program

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

Chapter 3. Aligning data

XL C/C++ provides many mechanisms for specifying data alignment at the levels of individual variables, members of aggregates, entire aggregates, and entire compilation units. If you are porting applications between different platforms, or between 32-bit and 64-bit modes, you will need to take into account the differences between alignment settings available in the different environments, to prevent possible data corruption and deterioration in performance. In particular, vector types have special alignment requirements which, if not followed, can produce incorrect results.

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

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

"Using alignment modifiers" on page 13 discusses the different specifiers, pragmas, and attributes you can use in your source code to override the alignment mode currently in effect, for specific variable declarations. It also provides the rules governing the precedence of alignment modes and modifiers during compilation.

Using alignment modes

Each data type supported by XL C/C++ is aligned along byte boundaries according to platform-specific default alignment *modes*, as follows:

- AIX **power** or **full**, which are equivalent.
- Linux **linuxppc**.

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

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

Set the alignment mode for all variables in a section of source code
To use this approach, you specify the **#pragma align** or **#pragma options align** directives in the source files, with one of the suboptions listed in Table 10 on page 10. Each directive changes the alignment mode in effect for all variables that follow the directive until another directive is encountered, or until the end of the compilation unit.

Each of the valid alignment modes is defined in Table 10 on page 10, which provides the alignment value, in bytes, for scalar variables, for all data types. For considerations of cross-platform compatibility, the table indicates the alignment values for each alignment mode on the UNIX® platforms. Where there are

differences between 32-bit and 64-bit modes, these are indicated. Also, where there are differences between the first (scalar) member of an aggregate and subsequent members of the aggregate, these are indicated.

Table 10. Alignment settings (values given in bytes)

Data type	Storage	Alignment settings and supported platforms					
		natural	**power, full**	**mac68k, twobyte**	**linuxppc**	**bit_packed**	**packed**
		AIX	AIX	AIX	Linux	AIX Linux	AIX
_Bool (C), bool (C++) (32-bit mode)	1	1	1	1	1	1	1
_Bool (C), bool (C++) (64-bit mode)	1	1	1	not supported	1	1	1
char, signed char, unsigned char	1	1	1	1	1	1	1
wchar_t (32-bit mode)	2	2	2	2	2	1	1
wchar_t (64-bit mode)	4	4	4	not supported	4	1	1
int, unsigned int	4	4	4	2	4	1	1
short int, unsigned short int	2	2	2	2	2	1	1
long int, unsigned long int (32-bit mode)	4	4	4	2	4	1	1
long int, unsigned long int (64-bit mode)	8	8	8	not supported	8	1	1
long long	8	8	8	2	8	1	1
float	4	4	4	2	4	1	1
double	8	8	see note[1]	2	8	1	1
long double	8	8	see note[1]	2	8	1	1
pointer (32-bit mode)	4	4	4	2	4	1	1
pointer (64-bit mode)	8	8	8	not supported	8	1	1
vector types	16	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.

If you generate data with an application on one platform and read the data with an application on another platform, it is recommended that you use the **bit_packed** mode, which results in equivalent data alignment on all platforms.

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

"Alignment of aggregates" on page 11 discusses the rules for the alignment of entire aggregates and provide examples of aggregate layouts. "Alignment of bit

fields" discusses additional rules and considerations for the use and alignment of bit fields, and provides an example of bit-packed alignment.

Related information

- **-qalign**,**#pragma align**, and **-qaltivec** in the *XL C/C++ Compiler Reference*

Alignment of aggregates

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

- For all alignment modes, the *size* of an aggregate is the smallest multiple of its alignment value that can encompass all of the members of the aggregate.
- C Empty aggregates are assigned a size of 0 bytes.
- C++ Empty aggregates are assigned a size of 1 byte. Note that static data members do not participate in the alignment or size of an aggregate; therefore a structure or class containing only a single static data member has a size of 1 byte.
- 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.
- 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.

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

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

Alignment of bit fields

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

C The length of a bit field cannot exceed the length of its base type. In extended mode, you can use the `sizeof` operator on a bit field. The `sizeof` operator on a bit field always returns the size of the base type.

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

However, alignment rules for aggregates containing bit fields are different depending on the alignment mode in effect. These rules are described below.

Rules for Linux PowerPC alignment

- Bit fields are allocated from a bit field container. The size of this container is determined by the declared type of the bit field. For example, a `char` bit field

uses an 8-bit container, an `int` bit field uses 32 bits, and so on. The container must be large enough to contain the bit field, as the bit field will not be split across containers.

- Containers are aligned in the aggregate as if they start on a natural boundary for that type of container. Bit fields are not necessarily allocated at the start of the container.
- If a zero-length bit field is the first member of an aggregate, it has no effect on the alignment of the aggregate and is overlapped by the next data member. If a zero-length bit field is a non-first member of the aggregate, it pads to the next alignment boundary determined by its base declared type but does not affect the alignment of the aggregate.
- Unnamed bit fields do not affect the alignment of the aggregate.

Rules for bit-packed alignment

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

Example of bit-packed alignment

For:

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

pragma options align=reset
```

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

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

Using alignment modifiers

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

#pragma pack(...)

Valid application:
The entire aggregate (as a whole) immediately following the directive.

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:
`1`, `2`, `4`, `8`, `16`, `nopack`, `pop`, and empty brackets. The use of empty brackets has the same functionality as `nopack`.

__attribute__((aligned(n)))

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

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

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

__attribute__((packed))

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

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

__align(n)

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

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

Valid values:

n must be a positive power of 2. XL C/C++ also allows you to specify a value greater than the system maximum, up to an absolute maximum of .

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 ″Type Attributes″ in the *XL C/C++ Language Reference*.

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

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

Related information

- ″The aligned variable attribute″, ″The packed variable attribute″, ″The aligned type attribute″, ″The packed type attribute″, and ″The __align specifier″ in the *XL C/C++ Language Reference*
- **#pragma pack** in the *XL C/C++ Compiler Reference*

Precedence rules for scalar variables

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

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

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

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

and where *type alignment* is the alignment mode currently in effect when the variable is declared, or the alignment value applied to a type in a `typedef` statement.

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

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

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

Precedence rules for aggregate variables

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

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

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

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

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

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

Chapter 4. Handling floating point operations

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

Floating-point formats

XL C/C++ supports the following floating-point formats:

- 32-bit single precision, with an approximate range of 10^{-38} to 10^{+38} and precision of about 7 decimal digits
- 64-bit double precision, with an approximate range of 10^{-308} to 10^{+308} and precision of about 16 decimal digits

Handling multiply-add operations

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

Related information

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

Compiling for strict IEEE conformance

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

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

Related information

- "Using optimization levels" on page 36
- **-qfloat** in the *XL C/C++ Compiler Reference*
- **-qstrict** in the *XL C/C++ Compiler Reference*
- **-qinitauto** in the *XL C/C++ Compiler Reference*

Handling floating-point constant folding and rounding

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

- The compile-time rounding mode is different from the execution-time rounding mode. By default, both are round-to-nearest; however, if your program changes the execution-time rounding mode, to avoid differing results, do either of the following:
  - Change the compile-time rounding mode to match the execution-time mode, by compiling with the appropriate **-y** option. For more information, and an example, see "Matching compile-time and runtime rounding modes."
  - Suppress folding, by compiling with the **-qfloat=nofold** option.
- Expressions like *a+b*c* are partially or fully evaluated at compile time. The results might be different from those produced at execution time, because *b*c* might be rounded before being added to *a*, while the runtime multiply-add instruction does not use any intermediate rounding. To avoid differing results, do either of the following:
  - Suppress the use of multiply-add instructions, by compiling with the **-qfloat=nomaf** option.
  - Suppress folding, by compiling with the **-qfloat=nofold** option.
- An operation produces an infinite or NaN result. Compile-time folding prevents execution-time detection of an exception, even if you compile with the **-qflttrap** option. To avoid missing these exceptions, suppress folding with the **-qfloat=nofold** option.

Related information

- "Handling floating-point exceptions" on page 19
- **-qfloat** and **-qstrict** in the *XL C/C++ Compiler Reference*

Matching compile-time and runtime rounding modes

The default rounding mode used at compile time and run time is round-to-nearest. 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:

```
#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 */
 printf ("1/3 with execution-time rounding to zero = %.17f\n", one_third);

fesetround (FE_TONEAREST);
one_third = one / three;  /* not folded */
 printf ("1/3 with execution-time rounding to nearest = %.17f\n", 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_DOWNWARD);
one_third = one / three;  /* not folded */
 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 compile-time rounding = 0.33333333333333331
1/3 with execution-time rounding to zero = 0.33333333333333331
1/3 with execution-time rounding to nearest   = 0.33333333333333331
1/3 with execution-time rounding to +infinity = 0.33333333333333337
1/3 with execution-time rounding to -infinity = 0.33333333333333331
```

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

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

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

Related information

- **-qfloat** and **-y** in the *XL C/C++ Compiler Reference*

Handling floating-point exceptions

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

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

Related information

- **-qfloat** and **-qflttrap** in the *XL C/C++ Compiler Reference*

Chapter 5. Using C++ templates

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

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

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

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

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

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

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

Have the compiler store instantiation information in a registry
Use the **-qtemplateregistry** compiler option. Information about each template instantiation is stored in a template registry. If the compiler is asked to instantiate the same template again with the same arguments, it points to the instantiation in the first object file instead. The **-qtemplateregistry** compiler option provides the benefits of the **-qtempinc** compiler option but does not require a specific structure for the template definition and implementation files. This approach is described in "Using the -qtemplateregistry compiler option" on page 24.

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

Using the -qtempinc compiler option

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

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

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

  This produces the following results:

  – Whenever you compile with **-qnotempinc**, the template implementation file is included.

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

Related information

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

Example of -qtempinc

This example includes the following source files:

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

In this example:

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

Template declaration file: stack.h

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

```
#ifndef STACK_H
#define STACK_H

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

```
      int   top;             // Index to top of stack
};

#ifndef __TEMPINC__          //  3
#include "stack.c"           //  3
#endif                       //  3
#endif
```

Template implementation file: stack.c

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

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

Function declaration file: stackops.h

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

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

Function implementation file: stackops.cpp

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

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

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

Main program file: stackadd.cpp

This file creates a `Stack` object.

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

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

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

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

  return(0);
    }
```

Regenerating the template instantiation file

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

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

Using -qtempinc with shared libraries

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

If you use **-qtempinc**:

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

Using the -qtemplateregistry compiler option

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

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

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

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

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

Related information

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

Recompiling related compilation units

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

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

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

Switching from -qtempinc to -qtemplateregistry

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

- If your application compiles successfully with both **-qtempinc** and **-qnotempinc**, you do not need to make any changes.
- If your application compiles successfully with **-qtempinc** but not with **-qnotempinc**, you must change it so that it will compile successfully with **-qnotempinc**. In each template definition file, conditionally include the corresponding template implementation file if the __TEMPINC__ macro is not defined. This is illustrated in "Example of -qtempinc" on page 22.

Chapter 6. Constructing a library

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

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

"Initializing static objects in libraries (C++)" on page 28 describes how to use priorities to control the order of initialization of objects across multiple files in a C++ application.

Compiling and linking a library

Compiling a static library

To compile a static library:

1. Compile each source file into an object file, with no linking. For example:
   ```
   xlc -c bar.c example.c
   ```
2. Use the Linux **ar** command to add the generated object files to an archive library file. For example:
   ```
   ar -rv libfoo.a bar.o example.o
   ```

Compiling a shared library

To compile a shared library:

1. Compile your source files into an object file, with no linking. For example:
   ```
   xlc -c foo.c
   ```
2. Use the **-qmkshrobj** compiler option to create a shared object from the generated object files. For example:
   ```
   xlc -qmkshrobj -o libfoo.so foo.o
   ```

Related information

- **-qmkshrobj** in the *XL C/C++ Compiler Reference*

Linking a library to an application

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

```
xlc -o myprogram main.c -Ldirectory [-Rdirectory] -lfoo
```

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

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

Linking a shared library to another shared library

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

```
xlc -qmkshrobj -o mylib.so myfile.o -Ldirectory -Rdirectory -lfoo
```

Related information

- **-qmkshrobj**, **-l**, **-R** and **-L** in the *XL C/C++ Compiler Reference*

Initializing static objects in libraries (C++)

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

To specify an initialization order for objects, you assign relative *priority* numbers to objects. The mechanisms by which you can specify priorities for entire files or objects within files are discussed in "Assigning priorities to objects." The mechanisms by which you can control the initialization order of objects across modules are discussed in "Order of object initialization across libraries" on page 30.

Assigning priorities to objects

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

Set the priority level for an entire file

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

Set the priority level for objects within a file

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

Linux **Set the priority level for individual objects**

To use this approach, you use `init_priority` variable attributes in the source files. The `init_priority` attribute takes precedence over **#pragma priority** directives, and can be applied to objects in any declaration order. On Linux, the objects are initialized according to their priority and terminated in reverse priority *across* compilation units.

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

Related information

- "The init_priority variable attribute" in the *XL C/C++ Language Reference*

Using priority numbers

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

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

The examples below show how to specify the priority of objects within a single file, and across two files. "Order of object initialization across libraries" on page 30 provides detailed information on the order of initialization of objects on the Linux platform.

Example of object initialization within a file

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

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

static Base a ;

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

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

Garage d ;
...
```

Example of object initialization across multiple files

The following example describes the initialization order for objects in two files, `farm.C` and `zoo.C`. Both files are contained in the same shared module, and use the **-qpriority** compiler option and **#pragma priority** directives.

farm.C -qpriority=1000

```
...
Dog a ;
Dog b ;
...
#pragma priority(6000)
...
Cat c ;
Cow d ;
...
#pragma priority(7000)
Mouse e ;
...
```

zoo.C -qpriority=2000

```
...

Bear m ;
...
#pragma priority(5000)
...
Zebra n ;
Snake s ;
...
#pragma priority(8000)
Frog f ;
...
```

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

Sequence	Object	Priority value	Comment
1	Dog a	1000	Takes option priority (1000).
2	Dog b	1000	Follows with the same priority.
3	Bear m	2000	Takes option priority (2000).
4	Zebra n	5000	Takes pragma priority (5000).
5	Snake s	5000	Follows with same priority.
6	Cat c	6000	Next priority number.
7	Cow d	6000	Follows with same priority.
8	Mouse e	7000	Next priority number.
9	Frog f	8000	Next priority number (initialized last).

Order of object initialization across libraries

Each static library and shared library is loaded and initialized at run time in *reverse* link order, once all of its dependencies have been loaded and initialized. Link order is the order in which each library was listed on the command line during linking into the main application. For example, if library A calls library B, library B is loaded before library A.

As each module is loaded, objects are initialized in order of priority, according to the rules outlined in "Assigning priorities to objects" on page 28. If objects do not have priorities assigned, or have the same priorities, object files are initialized in reverse link order — where link order is the order in which the files were given on the command line during linking into the library — and the objects within the files are initialized according to their declaration order. Objects are terminated in reverse order of their construction.

Example of object initialization across libraries

In this example, the following modules are used:

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

The source files are compiled into object files with the following command strings:

```
xlC -qpriority=101 -c fileA.C -o fileA.o
xlC -qpriority=150 -c fileB.C -o fileB.o
xlC -c fileC.C -o fileC.o
xlC -c fileD.C -o fileD.o
xlC -c fileE.C -o fileE.o
xlC -c fileF.C -o fileF.o
xlC -qpriority=300 -c fileG.C -o fileG.o
xlC -qpriority=200 -c fileH.C -o fileH.o
xlC -qpriority=500 -c fileI.C -o fileI.o
xlC -c fileJ.C -o fileJ.o
xlC -c fileK.C -o fileK.o
xlC -qpriority=600 -c fileL.C -o fileL.o
```

The dependent libraries are created with the following command strings:

```
xlC -qmkshrobj -o libS3.so fileE.o fileF.o
xlC -qmkshrobj -o libS4.so fileG.o fileH.o
xlC -qmkshrobj -o libS5.so fileI.o fileJ.o
xlC -qmkshrobj -o libS6.so fileK.o fileL.o
```

The dependent libraries are linked with their parent libraries using the following command strings:

```
xlC -qmkshrobj -o libS1.so fileA.o fileB.o -L. -R. -lS3 -lS4
xlC -qmkshrobj -o libS2.so fileC.o fileD.o -L. -R. -lS5 -lS6
```

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

```
xlC main.C -o main.out -L. -R. -lS1 -lS2
```

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

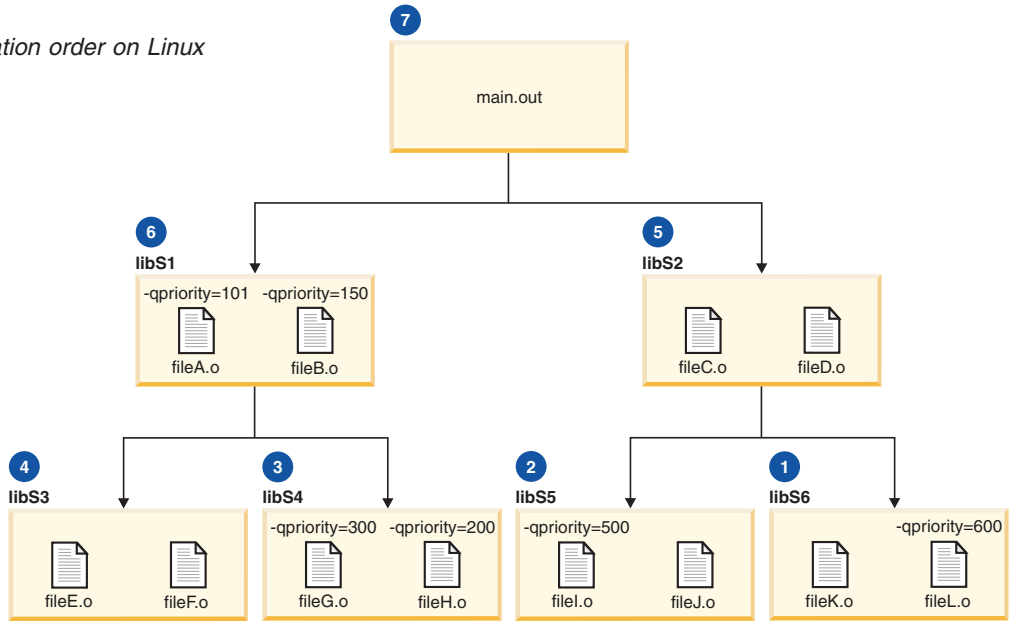


Figure 1. Object initialization order on Linux

Objects are initialized as follows:

Sequence	Object	Priority value	Comment
1	libS6	n/a	libS2 was entered last on the command line when linked with main, and so is initialized before libS1. However, libS5 and libS6 are dependencies of libS2, so they are initialized first. Since it was entered last on the command line when linked to create libS2, libS6 is initialized first. The objects in this library are initialized according to their priority.
2	fileL	600	The objects in fileL are initialized next (lowest priority number in this module).
3	fileK	65535	The objects in fileK are initialized next (next priority number in this module (default priority of 65535)).
4	libS5	n/a	libS5 was entered before libS6 on the command line when linked with libS2, so it is initialized next. The objects in this library are initialized according to their priority.
5	fileI	500	The objects in fileI are initialized next (lowest priority number in this module).
6	fileJ	65535	The objects in fileJ are initialized next (next priority number in this module (default priority of 65535)).
7	libS4	n/a	libS4 is a dependency of libS1 and was entered last on the command line when linked to create libS1, so it is initialized next. The objects in this library are initialized according to their priority.
8	fileH	200	The objects in fileH are initialized next (lowest priority number in this module).
9	fileG	300	The objects in fileG are initialized next (next priority number in this module).
10	libS3	n/a	libS3 is a dependency of libS1 and was entered first on the command line during the linking with libS1, so it is initialized next. The objects in this library are initialized according to their priority.
11	fileF	65535	Both fileF and fileE are assigned a default priority of 65535. However, because fileF was listed last on the command line when the object files were linked into libS3, fileF is initialized first.
12	fileE	65535	Initialized next.
13	libS2	n/a	libS2 is initialized next. The objects in this library are initialized according to their priority.
14	fileD	65535	Both fileD and fileC are assigned a default priority of 65535. However, because fileD was listed last on the command line when the object files were linked into libS2, fileD is initialized first.
15	fileC	65535	Initialized next.
16	libS1		libS1 is initialized next. The objects in this library are initialized according to their priority.
17	fileA	101	The objects in fileA are initialized next (lowest priority number in this module).

Sequence	Object	Priority value	Comment
18	fileB	150	The objects in `fileB` are initialized next (next priority number in this module).
19	main.out	n/a	Initialized last. The objects in `main.out` are initialized according to their priority.

Chapter 7. Optimizing your applications

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

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

- You can use an option that performs a specific type of optimization, including:
  - System architecture. If your application will run on a specific hardware configuration, the compiler can generate instructions that are optimized for the target machine, including microprocessor architecture, cache or memory geometry, and addressing model. These options are discussed in "Optimizing for system architecture" on page 39.
  - High-order loop analysis and transformation. The compiler uses various techniques to optimize loops. These options are discussed in "Using high-order loop analysis and transformations" on page 40.
  - "Using shared-memory parallelism (SMP)" on page 41. If your application will run on hardware that supports shared memory parallelization, you can instruct the compiler to automatically generate threaded code, or to recognize OpenMP standard programming constructs. Options for parallelizing your program are discussed in "Using shared-memory parallelism (SMP)" on page 41.
  - Interprocedural analysis (IPA). The compiler reorganizes code sections to optimize calls between functions. IPA options are discussed in "Using interprocedural analysis" on page 42.
  - Profile-directed feedback (PDF). The compiler can optimize sections of your code based on call and block counts and execution times. PDF options are discussed in "Using profile-directed feedback" on page 44
  - Other types of optimization, including loop unrolling, function inlining, stack storage compacting, and many others. Brief descriptions of these options are provided in "Other optimization options" on page 47.
- You can use an optimization *level*, which bundles several techniques and may include one or more of the aforementioned specific optimization options. There are four optimization levels that perform increasingly aggressive optimizations on your code. Optimization levels are described in "Using optimization levels" on page 36.
- You can combine optimization options and levels to achieve the precise results you want. Discussions on how to do so are provided throughout the sections referenced above.

Keep in mind that program optimization implies a trade-off, in that it results in longer compile times, increased program size and disk usage, and diminished debugging capability. At higher levels of optimization, program semantics might be affected, and code that executed correctly before optimization might no longer run

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

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

Finally, if you want to manually tune your application to complement the optimization techniques used by the compiler, Chapter 8, “Coding your application to improve performance,” on page 49 provides suggestions and best practices for coding for performance.

Related information

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

Using optimization levels

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

Table 11. Optimization levels

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

Techniques used in optimization level 2

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

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

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

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

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

Techniques used in optimization level 3

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

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

"Getting the most out of optimization levels 2 and 3" on page 38 provides some suggestions for mitigating this risk.

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

- Unrolling deeper loops and improving loop scheduling.
- Increasing the scope of optimization.
- Performing optimizations with marginal or niche effectiveness, which might not help all programs.
- Performing optimizations that are expensive in compile time or space.
- Reordering some floating-point computations, which might produce precision differences or affect the generation of floating-point-related exceptions (equivalent to compiling with the **-qnostrict** option).
- Eliminating implicit memory usage limits (equivalent to compiling with the **-qmaxmem=-1** option).
- Performing a subset of high-order transformations (equivalent to compiling with the **-qhot=level=0** option).
- Increasing automatic inlining.
- Propagating constants and values through structure copies.

- Removing the ″address taken″ attribute if possible after other optimizations.
- Grouping loads, stores and other operations on contiguous aggregate members, in some cases using VMX vector register operations.

Techniques used in optimization levels 4 and 5

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

- High-order transformations, which provide optimized handling of loop nests (equivalent to compiling with the **-qhot=level=1** option).
- Interprocedural analysis, which invokes the optimizer at link time to perform optimizations across multiple source files (equivalent to compiling with the **-qipa** option).
- Hardware-specific optimization (equivalent to compiling with the **-qarch=auto**, **-qtune=auto**, and **-qcache=auto** options).
- At optimization level 5, more detailed interprocedural analysis (the equivalent to compiling with the **-qipa=level=2** option). With level 2 IPA, high-order transformations are delayed until link time, after whole-program information has been collected.

Getting the most out of optimization levels 2 and 3

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

1. If possible, test and debug your code without optimization before using **-O2**.
2. Ensure that your code complies with its language standard.
3. C In C code, ensure that the use of pointers follows the type restrictions: generic pointers should be `char*` or `void*`. Also check that all shared variables and pointers to shared variables are marked `volatile`.
4. C In C, use the **-qlibansi** compiler option unless your program defines its own functions with the same names as library functions.
5. Compile as much of your code as possible with **-O2**.
6. If you encounter problems with **-O2**, consider using **-qalias=noansi** rather than turning off optimization.
7. Next, use **-O3** on as much code as possible.
8. If your application is sensitive to floating-point exceptions or the order of evaluation for floating-point arithmetic, use **-qstrict** along with **-O3** to ensure accurate results, while still gaining most of the performance benefits of **-O3**.
9. If you encounter unacceptably large code size, try using **-qcompact** along with **-O3** where necessary.
10. If you encounter unacceptably long compile times, consider disabling the high-order transformations by using **-qnohot**.
11. If you still have problems with **-O3**, switch to **-O2** for a subset of files, but consider using **-qmaxmem=-1**, **-qnostrict**, or both.

Related information

- **-qstrict**, **-qmaxmem**, **-qunroll**, and **-qalias**, in the *XL C/C++ Compiler Reference*

Optimizing for system architecture

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

Table 12. Target machine options

Option	Behavior
-q32	Generates code for a 32-bit (4/4/4) addressing model (32-bit execution mode). This is the default setting.
-q64	Generates code for a 64-bit (4/8/8) 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. The default on all Linux distributions is **-qarch=ppc64grsq**. Using **-O4** or **-O5** sets the default to **-qarch=auto**. See "Getting the most out of target machine options" below for more information on this option.
-qipa=clonearch	Allows you to specify multiple specific processor architectures for which instruction sets will be generated. At run time, the application will detect the specific architecture of the operating environment and select the instruction set specialized for that architecture. The advantage of this option is that it allows you to optimize for several architectures without recompiling your code for each target architecture. See "Using interprocedural analysis" on page 42 for more information on this option.
-qtune	Biases optimization toward execution on a given microprocessor, without implying anything about the instruction set architecture to use as a target. The default is **-qtune=pwr4**. See "Getting the most out of target machine options" below for more information on this option.
-qcache	Defines a specific cache or memory geometry. The defaults are determined through the setting of **-qtune**. See "Getting the most out of target machine options" below for more information on this option.

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

Getting the most out of target machine options

Using -qarch options

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

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

Using -qtune options

If you specify a particular architecture with **-qarch**, **-qtune** will automatically select the suboption that generates instruction sequences with the best performance for that architecture. If you specify a *group* of architectures with **-qarch**, compiling with **-qtune=auto** will generate code that runs on all of the architectures in the specified group, but the instruction sequences will be those with the best performance on the architecture of the compiling machine.

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

Using -qcache options

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

Related information

- "Using the Mathematical Acceleration Subsystem (MASS)" on page 55
- **-qarch**, **-qcache**, **-qtune**, and **-qlistopt** in the *XL C/C++ Compiler Reference*

Using high-order loop analysis and transformations

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

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

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

Table 13. **-qhot** *suboptions*

suboption	Behavior
level=1	This is the default suboption if you specify **-qhot** with no suboptions. This level is also automatically enabled if you compile with **-O4** or **-O5**. This is equivalent to specifying **-qhot=vector** and, where applicable, **-qhot=simd**.
level=0	Instructs the compiler to perform a subset of high-order transformations that enhance performance by improving data locality. This suboption implies **-qhot=novector**, **-qhot=noarraypad**, and **-qhot=nosimd**. This level is automatically enabled if you compile with **-O3**.

Table 13. **-qhot** *suboptions (continued)*

suboption	Behavior
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.
arraypad	Instructs the compiler to pad any arrays where it infers there might be a benefit and to pad by whatever amount it chooses.
simd	Instructs the compiler to attempt automatic SIMD vectorization; that is, converting certain operations in a loop that apply to successive elements of an array into a call to a VMX instruction. This call calculates several results at one time, which is faster than calculating each result sequentially. This suboption is enabled by default on Linux if you set **-qarch** to a target architecture that supports VMX instructions (and **-qenablevmx** is in effect, which it is by default).

Getting the most out of -qhot

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

- Try using **-qhot** along with **-O3** for all of your code. It is designed to have a neutral effect when no opportunities for transformation exist.
- If the runtime performance of your code can significantly benefit from automatic inlining and memory locality optimizations, try using **-O4** with **-qhot=level=0** or **-qhot=novector**.
- If you encounter unacceptably long compile times (this can happen with complex loop nests), use **-qhot=level=0**.
- If your code size is unacceptably large, try using **-qcompact** along with **-qhot**.
- If necessary, deactivate **-qhot** selectively, allowing it to improve some of your code.

Related information

- **-qhot**, **-qenablevmx**, and **-qstrict** in *XL C/C++ Compiler Reference*

Using shared-memory parallelism (SMP)

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

The following table lists the most commonly used suboptions. Descriptions and syntax of all the suboptions are provided in **-qsmp** in the *XL C/C++ Compiler Reference*. An overview of automatic parallelization, as well as of OpenMP directives is provided in Chapter 10, “Parallelizing your programs,” on page 65.

Table 14. 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 OpenMP directives, are also recognized. This is the default setting if you do not specify any **-qsmp** suboptions, and it also implies the **opt** suboption.
omp	Instructs the compiler to enforce strict conformance to the OpenMP API for specifying explicit parallelism. Only language constructs that conform to the OpenMP standard are recognized. Note that **-qsmp=omp** is currently incompatible with **-qsmp=auto**.
opt	Instructs the compiler to optimize as well as parallelize. The optimization is equivalent to **-O2 -qhot** in the absence of other optimization options.
fine_tuning	Other values for the suboption provide control over thread scheduling, nested parallelism, locking, etc.

Getting the most out of -qsmp

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

- Before using **-qsmp** with automatic parallelization, test your programs using optimization and **-qhot** in a single-threaded manner.
- If you are compiling an OpenMP program and do not want automatic parallelization, use **-qsmp=omp:noauto** .
- Always use the reentrant compiler invocations (the *_r* invocations) when using **-qsmp**.
- By default, the 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

- "Environment variables for parallel processing" in *XL C/C++ Compiler Reference*
- "Invoking the compiler" in *XL C/C++ Compiler Reference*

Using interprocedural analysis

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

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

Table 15. 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, code motion into conditions or out of loops, 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=*variable*	Allows precise control over function inlining.
clonearch=*arch_list*	Allows you to specify multiple architectures for which optimized instructions can be generated. Supported architecture values are **PWR4**, **PWR5**, and **PPC970**. For every function in your program, the compiler generates a generic version of the instruction set, according to the **-qarch** value in effect, and, if appropriate, *clones* specialized versions of the instruction set for the architectures you specify in this suboption. The compiler inserts code into your application to check for the processor architecture at run time, and selects the version of the generated instructions that is optimized for the runtime environment.
cloneproc=*func_list*	Allows you to specify the exact functions which should be cloned for the specified architectures in the **clonearch** suboption.
fine_tuning	Other values for **-qipa** provide the ability to specify the behavior of library code, tune program partitioning, read commands from a file, etc.

Getting the most from -qipa

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

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

Note: While IPA's interprocedural optimizations can significantly improve performance of a program, they can also cause previously incorrect but functioning programs to fail. Some programming practices that can work by accident without aggressive optimization but are exposed with IPA include:

- Relying on the allocation order or location of automatic variables, such as taking the address of an automatic variable and then later comparing it with the address of another local variable to determine the growth direction of a stack. The C language does not guarantee where an automatic variable is allocated, or its position relative to other automatic variables. Do not compile such a function with IPA.
- Accessing a pointer that is either invalid or beyond an array's bounds. Because IPA can reorganize global data structures, a wayward pointer which might have previously modified unused memory might now trample upon user-allocated storage.

Related information

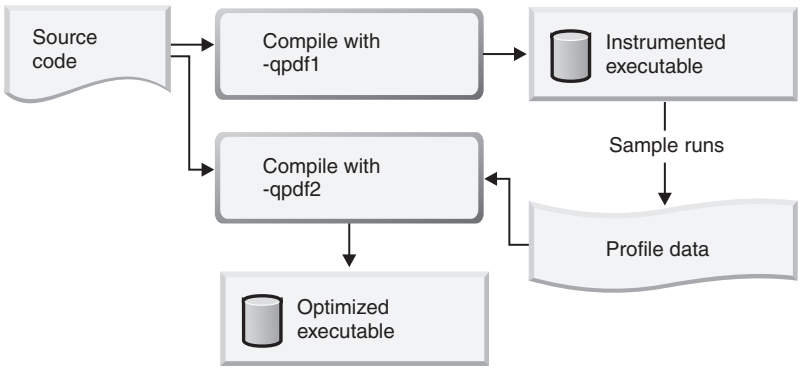
- **-qipa**, **-Q**, and **-qlist** in the *XL C/C++ Compiler Reference*

Using profile-directed feedback

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

The following diagram illustrates the PDF process.

Figure 2. Profile-directed feedback



You first compile the program with the **-qpdf1** option (with a minimum optimization level of **-O**), which generates profile data by using the compiled program in the same ways that users will typically use it. You then compile the program again, with the **-qpdf2** option. This optimizes the program based on the profile data, by invoking **-qipa=level=0**. Alternatively, if you want to save considerable time by avoiding a full re-compilation in the **-qpdf2** step, you can simply re-link the object files produced by the **-qpdf1** step.

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

To use the **-qpdf** options:

1. Compile some or all of the source files in the application with **-qpdf1** and a minimum of **-O**. If you want to avoid full re-compilation in the **-qpdf2** step, first compile with **-qpdf1** but with no linking, so that the object files are saved. Then link the object files into an executable. To use this procedure, issue commands similar to the following example:

   ```
   xlc -c -qpdf1 -O file1.c file2.c
   xlc -qpdf1 -O file1.o file2.o
   ```

2. Run the application using a typical data set or several typical data sets. It is important to use data that is representative of the data that will be used by your application in a real-world scenario. When the application exits, it writes profiling information to the PDF file in the current working directory or the directory specified by the PDFDIR environment variable.
3. Re-compile the application with **-qpdf2** and a minimum of **-O**. Alternatively, if you saved the object files in step 1, you can simply re-link the object files; for example, issue the following command:

   ```
   xlc -qpdf2 -O file1.o file2.o
   ```

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

1. Compile some or all of the source files in the application with **-qpdf1** and a minimum of **-O**.
2. Run the application using a typical data set or several typical data sets. This produces a PDF file in the current directory.
3. Change the directory specified by the PDFDIR environment variable to produce a PDF file in a different directory.
4. Re-compile or re-link the application with **-qpdf1** and a minimum of **-O**.
5. Repeat steps 3 and 4 as often as you want.

6. Use the **mergepdf** utility to combine the PDF files into one PDF file. For example, if you produce three PDF files that represent usage patterns that will occur 53%, 32%, and 15% of the time respectively, you can use this command:
   ```
   mergepdf -r 53 path1  -r 32 path2  -r 15 path3
   ```
7. Re-compile or or re-link the application with **-qpdf2** and a minimum of **-O**.

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

1. Compile the application with **-qpdf1 -qshowpdf -O**.
2. Run the application using a typical data set or several typical data sets. The application writes more detailed profiling information in the PDF file.
3. Use the **showpdf** utility to view the information in the PDF file.

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

Example of compilation with pdf and showpdf

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

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

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

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

The results will look similar to the following:

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

Call Counters:
5 | 1  printf(6)

Call coverage = 100% ( 1/1 )

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

Block coverage = 100% ( 2/2 )

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

Call Counters:
10 | 1  HelloWorld(4)
```

```
Call coverage = 100% ( 1/1 )

Block Counters:
8-11 | 1
11 |

Block coverage = 100% ( 1/1 )

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

Related information

- **-qpdf** and **-showpdf** in the *XL C/C++ Compiler Reference*

Other optimization options

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

Table 16. Selected compiler options for optimizing performance

Option	Description
-qignerrno	Allows the compiler to assume that `errno` is not modified by library function calls, so that such calls can be optimized. Also allows optimization of square root operations, by generating inline code rather than calling a library function. (For processors that support `sqrt`.)
-qsmallstack	Instructs the compiler to compact stack storage. Doing so may increase heap usage.
-qinline	Controls inlining of named functions. Can be used at compile time, link time, or both. When **-qipa** is used, **-qinline** is synonymous with **-qipa=inline**.
-qunroll	Independently controls loop unrolling. 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. 64-bit mode only.
-qtbtable	Controls the generation of traceback table information. 64-bit mode only.
C++ **-qnoeh**	Informs the compiler that no C++ exceptions will be thrown and that cleanup code can be omitted. If your program does not throw any C++ exceptions, use this option to compact your program by removing exception-handling code.
-qnounwind	Informs the compiler that the stack will not be unwound while any routine in this compilation is active. This option can improve optimization of non-volatile register saves and restores. In C++, the **-qnounwind** option implies the **-qnoeh** option.

Chapter 8. Coding your application to improve performance

Chapter 7, "Optimizing your applications," on page 35 discusses the various compiler options that XL C/C++ provides for optimizing your code with minimal coding effort. If you want to take your application a step further, to complement and take the most advantage of compiler optimizations, the following sections discuss C and C++ programming techniques that can improve performance of your code:

Find faster input/output techniques

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

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

Reduce function-call overhead

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

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

- Use constant arguments in inlined functions whenever possible. Functions with constant arguments provide more opportunities for optimization.
- Use the **#pragma isolated_call** preprocessor directive to list functions that have no side effects and do not depend on side effects.
- Use **#pragma disjoint** within functions for pointers or reference parameters that can never point to the same memory.
- Declare a nonmember function as static whenever possible. This can speed up calls to the function.
- C++ Usually, you should not declare virtual functions inline. If all virtual functions in a class are inline, the virtual function table and all the virtual function bodies will be replicated in each compilation unit that uses the class.
- C++ When declaring functions, use the `const` specifier whenever possible.
- C 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.
- C Avoid using unprototyped variable argument functions.
- Design functions so that the most frequently used parameters are in the leftmost positions in the function prototype.
- Avoid passing by value structures or unions as function parameters or returning a structure or a union. Passing such aggregates requires the compiler to copy and store many values. This is worse in C++ programs in which class objects are passed by value because a constructor and destructor are called when the function is called. Instead, pass or return a pointer to the structure or union, or pass it by reference.
- Pass non-aggregate types such as `int` and `short` by value rather than passing by reference, whenever possible.
- If your function exits by returning the value of another function with the same parameters that were passed to your function, put the parameters in the same order in the function prototypes. The compiler can then branch directly to the other function.
- Use the built-in functions, which include string manipulation, floating-point, and trigonometric functions, instead of coding your own. Intrinsic functions require less overhead and are faster than a function call, and often allow the compiler to perform better optimization.

  C++ Your functions are automatically mapped to built-in functions if you include the XL C/C++ header files.

  C Your functions are mapped to built-in functions if you include `math.h` and `string.h`.
- Selectively mark your functions for inlining, using the `inline` keyword. An inlined function requires less overhead and is generally faster than a function call. The best candidates for inlining are small functions that are called frequently from a few places, or functions called with one or more compile-time constant parameters, especially those that affect `if`, `switch` or `for` statements. You might also want to put these functions into header files, which allows automatic inlining across file boundaries even at low optimization levels. Be sure to inline all functions that only load or store a value, or use simple operators such as comparison or arithmetic operators. Large functions and functions that are called rarely might not be good candidates for inlining.

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

Related information

- **#pragma isolated_call**, **#pragma disjoint**, and **-qipa** in the *XL C/C++ Compiler Reference*

Manage memory efficiently

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

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

Optimize variables

Consider the following guidelines:

- Use local variables, preferably automatic variables, as much as possible.

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

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

Related information

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

Manipulate strings efficiently

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

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

Related information

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

Optimize expressions and program logic

Consider the following guidelines:

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

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

  When you must use mixed-mode arithmetic, code the integer and floating-point arithmetic in separate computations whenever possible.
- Avoid `goto` statements that jump into the middle of loops. Such statements inhibit certain optimizations.
- Improve the predictability of your code by making the fall-through path more probable. Code such as:

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

  should be written as:

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

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

Optimize operations in 64-bit mode

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

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

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

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

- Avoid long division whenever possible. Multiplication is usually faster than division. If you need to perform many divisions with the same divisor, assign the reciprocal of the divisor to a temporary variable, and change all divisions to multiplications with the temporary variable. For example, the function

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

  will perform faster than the more straightforward:

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

  The reason is that the division `(1.0 / 1.0825)` is evaluated once, and folded, at compile time only.
- Use `long` types instead of `signed`, `unsigned`, and plain `int` types for variables which will be frequently accessed, such as loop counters and array indexes. Doing so frees the compiler from having to truncate or sign-extend array references, parameters during function calls, and function results during returns.

Chapter 9. Using the high performance libraries

XL C/C++ Advanced Edition for Linux 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 (MASS)."
- The Basic Linear Algebra Subprograms (BLAS) are a set of routines which provide matrix/vector multiplication functions tuned for PowerPC architectures. The BLAS functions are described in "Using the Basic Linear Algebra Subprograms (BLAS)" on page 62.

Using the Mathematical Acceleration Subsystem (MASS)

The MASS libraries consist of a library of scalar functions, described in "Using the scalar library"; and a set of vector libraries tuned for specific architectures, described in "Using the vector libraries" on page 57. The functions contained in both scalar and vector libraries are automatically called at certain levels of optimization, but you can also call them explicitly in your programs. Note that the accuracy and exception handling might not be identical in MASS functions and system library functions.

"Compiling and linking a program with MASS" on page 61 describes how to compile and link a program that uses the MASS libraries, and how to selectively use the MASS scalar library functions in concert with the regular `libm.a` scalar functions.

Using the scalar library

The MASS scalar library, `libmass.a`[1], contains an accelerated set of frequently used math intrinsic functions in the Linux math library. When you compile programs with any of the following options:

- **-qhot -qignerrno -qnostrict**
- **-qhot -O3**
- **-qsmp -qignerrno -qnostrict**
- **-qsmp -O3**
- **-O4**
- **-O5**

the compiler automatically uses the faster MASS functions for all math library functions (with the exception of `atan2`, `dnint`, `sqrt`, `rsqrt`). In fact, the compiler first tries to ″vectorize″ calls to math library functions by replacing them with the equivalent MASS vector functions; if it cannot do so, it uses the MASS scalar functions. When the compiler performs this automatic replacement of math library functions, it uses versions of the MASS functions contained in the system library `libxlopt.a`; you do not need to add any special calls to the MASS functions in your code, or to link to the `libxlopt` library.

Notes:

1. On Linux, 32-bit and 64-bit objects cannot be combined in the same library, so two versions of the scalar library are shipped with the compiler: `libmass.a` for 32-bit applications, and `libmass_64.a` for 64-bit applications.

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

1. Providing the prototypes for the functions (except `dnint` and `rsqrt`), by including `math.h` in your source files.
2. Providing the prototypes for `dnint` and `rsqrt`, by including `mass.h` in your source files.
3. Linking the MASS scalar library `libmass.a` (or the 64-bit version, `libmass_64.a`) with your application. For instructions, see "Compiling and linking a program with MASS" on page 61.

The MASS scalar functions accept double-precision parameters and return a double-precision result, and are summarized in Table 17.

Table 17. MASS scalar functions

Function	Description	Prototype
sqrt	Returns the square root of `x`	double sqrt (double x);
rsqrt	Returns the reciprocal of the square root of `x`	double rsqrt (double x);
exp	Returns the exponential function of `x`	double exp (double x);
expm1	Returns (the exponential function of `x`) – 1	double expm1 (double x);
log	Returns the natural logarithm of `x`	double log (double x);
log1p	Returns the natural logarithm of (`x` + 1)	double log1p (double x);
sin	Returns the sine of `x`	double sin (double x);
cos	Returns the cosine of `x`	double cos (double x);
tan	Returns the tangent of `x`	double tan (double x);
atan	Returns the arctangent of `x`	double atan (double x);
atan2	Returns the arctangent of `x/y`	double atan2 (double x, double y);
sinh	Returns the hyperbolic sine of `x`	double sinh (double x);
cosh	Returns the hyperbolic cosine of `x`	double cosh (double x);
tanh	Returns the hyperbolic tangent of `x`	double tanh (double x);
dnint	Returns the nearest integer to `x` (as a double)	double dnint (double x);
pow	Returns `x` raised to the power `y`	double pow (double x, double y);

The trigonometric functions (`sin`, `cos`, `tan`) return NaN (Not-a-Number) for large arguments (abs(x)>2**50*pi).

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

Using the vector libraries

When you compile programs with any of the following options:

- **-qhot -qignerrno -qnostrict**
- **-qhot -O3**
- **-qsmp -qignerrno -qnostrict**
- **-qsmp -O3**
- **-O4**
- **-O5**

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

If you are not using any of the optimization options listed above, and/or want to explicitly call any of the MASS vector functions, you can do so by including the header `massv.h` in your source files and linking your application with any of the following vector library archives (information on linking is provided in "Compiling and linking a program with MASS" on page 61):

libmassvp4.a
: Contains functions that have been tuned for the POWER4 architecture. If you are using a PPC970 machine, this library is the recommended choice.

libmassvp5.a
: Contains functions that have been tuned for the POWER5 architecture.

On Linux, 32-bit and 64-bit objects must not be mixed in a single library, so a separate 64-bit version of each vector library is provided: `libmassvp4_64.a`, and `libmassvp5_64.a`.

The single-precision and double-precision floating-point functions contained in the vector libraries are summarized in Table 18 on page 58. The integer functions contained in the vector libraries are summarized in Table 19 on page 60. Note that in C and C++ applications, only call by reference is supported, even for scalar arguments.

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

- a double-precision (for double-precision functions) or single-precision (for single-precision functions) vector output parameter
- a double-precision (for double-precision functions) or single-precision (for single-precision functions) vector input parameter
- an integer vector-length parameter

The functions are of the form *function_name* (*y*,*x*,*n*), where *y* is the target vector, *x* is the source vector, and *n* is the vector length. The parameters *y* and *x* are assumed to be double-precision for functions with the prefix `v`, and single-precision for functions with the prefix `vs`. As 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 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 18. 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 arccosine of x[i], for i=0,..,*n–1	void vacos (double y[], double x[], int *n);	void vsacos (float y[], float x[], int *n);
vasin	vsasin	Sets y[i] to the arcsine of x[i], for i=0,..,*n–1	void vasin (double y[], double x[], int *n);	void vsasin (float y[], float x[], int *n);
vatan2	vsatan2	Sets z[i] to the arctangent of x[i]/y[i], for i=0,..,*n–1	void vatan2 (double z[], double x[], double y[], int *n);	void vsatan2 (float z[], float x[], float y[], int *n);
vcbrt	vscbrt	Sets y[i] to the cube root of x[i], for i=0,..,*n-1	void vcbrt (double y[], double x[], int *n);	void vscbrt (float y[], float x[], int *n);
vcos	vscos	Sets y[i] to the cosine of x[i], for i=0,..,*n–1	void vcos (double y[], double x[], int *n);	void vscos (float y[], float x[], int *n);
vcosh	vscosh	Sets y[i] to the hyperbolic cosine of x[i], for i=0,..,*n–1	void vcosh (double y[], double x[], int *n);	void vscosh (float y[], float x[], int *n);
vcosisin	vscosisin	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);	
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);

Table 18. MASS floating-point vector functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
vexpm1	vsexpm1	Sets y[i] to (the exponential function of x[i])-1, for i=0,..,*n–1	void vexpm1 (double y[], double x[], int *n);	void vsexpm1 (float y[], float x[], int *n);
vlog	vslog	Sets y[i] to the natural logarithm of x[i], for i=0,..,*n–1	void vlog (double y[], double x[], int *n);	void vslog (float y[], float x[], int *n);
vlog10	vslog10	Sets y[i] to the base-10 logarithm of x[i], for i=0,..,*n–1	void vlog10 (double y[], double x[], int *n);	void vslog10 (float y[], float x[], int *n);
vlog1p	vslog1p	Sets y[i] to the natural logarithm of (x[i]+1), for i=0,..,*n–1	void vlog1p (double y[], double x[], int *n);	void vslog1p (float y[], float x[], int *n);
vpow	vspow	Sets z[i] to x[i] raised to the power y[i], for i=0,..,*n-1	void vpow (double z[], double x[], double y[], int *n);	void vspow (float z[], float x[], float y[], int *n);
vqdrt	vsqdrt	Sets y[i] to the fourth root of x[i], for i=0,..,*n-1	void vqdrt (double y[], double x[], int *n);	void vsqdrt (float y[], float x[], int *n);
vrcbrt	vsrcbrt	Sets y[i] to the reciprocal of the cube root of x[i], for i=0,..,*n-1	void vrcbrt (double y[], double x[], int *n);	void vsrcbrt (float y[], float x[], int *n);
vrec	vsrec	Sets y[i] to the reciprocal of x[i], for i=0,..,*n–1	void vrec (double y[], double x[], int *n);	void vsrec (float y[], float x[], int *n);
vrqdrt	vsrqdrt	Sets y[i] to the reciprocal of the fourth root of x[i], for i=0,..,*n-1	void vrqdrt (double y[], double x[], int *n);	void vsrqdrt (float y[], float x[], int *n);
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);

Table 18. MASS floating-point vector functions (continued)

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

Table 19. 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 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 vector of 64-bit objects	unsigned int vpopcnt8 (void *x, int *n)

The functions `vdiv`, `vsincos`, `vpow`, and `vatan2` (and their single-precision versions, `vsdiv`, `vssincos`, `vspow`, and `vsatan2`) take four parameters. The functions `vdiv`, `vpow`, and `vatan2` take the parameters (*z*,*x*,*y*,*n*). The function `vdiv` outputs a vector *z* whose elements are x[i]/y[i], where i=0,..,*n–1. The function `vpow` outputs a vector *z* whose elements are $x[i]^{y[i]}$, where i=0,..,*n–1. The function `vatan2` outputs a vector *z* whose elements are atan(x[i]/y[i]), where i=0,..,*n–1. The function `vsincos` takes the parameters (*y*,*z*,*x*,*n*), and outputs two vectors, *y* and *z*, whose elements are sin(x[i]) and cos(x[i]), respectively.

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)`). Other kinds of overlap (where input and output vectors are neither disjoint nor identical) should be avoided, since they may produce unexpected results:

- 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 unexpected results may be obtained.

- For calls to vector functions that take two input vectors (for example, `vatan2 (y, x1, x2, &n)`):

  The previous restriction applies to both pairs of vectors `y`, `x1` and `y`, `x2`. That is, `y[0:n-1]` and `x1[0:n-1]` must be either disjoint or identical; and `y[0:n-1]` and `x2[0:n-1]` must be either disjoint or identical.

- For calls to vector functions that take two output vectors (for example, `vsincos (y1, y2, x, &n)`):

  The above restriction applies to both pairs of vectors `y1, x` and `y2, x`. That is, `y1[0:n-1]` and `x[0:n-1]` must be either disjoint or identical; and `y2[0:n-1]` and `x[0:n-1]` must be either disjoint or identical. Also, the vectors `y1[0:n-1]` and `y2[0:n-1]` must be disjoint.

Consistency of MASS vector functions

All the functions in the MASS vector libraries are consistent, in the sense that a given input value will always produce the same result, regardless of its position in the vector, and regardless of the vector length.

Compiling and linking a program with MASS

To compile an application that calls the functions in the MASS libraries, specify **mass** and **massvp4** (or **massvp5**) (32-bit), or **mass_64** and **massvp4_64** (or **massvp5_64**) (64-bit) on the **-l** linker option. For example, if the MASS libraries are installed in the default directory, you could specify one of the following:

```
xlc progc.c -o progc -lmass -lmassvp4
xlc progc.c -o progc -lmass_64 -lmassvp4_64 -q64
```

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

Using libmass.a with libm.a

If you wish to use the `libmass.a` (or `libmass_64.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. Use the Linux **ar** command to extract the object files of the desired functions from `libmass.a` or `libmass_64.a`. For most functions, the object file name is the function name followed by `.s32.o` (for 32-bit mode) or `.s64.o` (for 64-bit mode).[1] For example, to extract the object file for the `tan` function in 32-bit mode, the command would be:

   ```
   ar -x tan.s32.o libmass.a
   ```

2. Archive the extracted object files into another library:

   ```
   ar -qv libfasttan.a tan.s32.o
   ranlib libfasttan.a
   ```

3. Create the final executable using **xlc**, specifying **-lfasttan** instead of **-lmass**:

   ```
   xlc sample.c -o sample dir_containing_libfasttan.a -lfasttan
   ```

   This links only the `tan` function from MASS (now in `libfasttan.a`) and the remainder of the math functions from the standard system library.

Notes:

1. The exceptions are:
   - The `sin` and `cos` functions are both contained in the object files `sincos.s32.o` and `sincos.s64.o`.
   - The `pow` function is contained in the object files `dxy.s32.o` and `dxy.s64.o`.
2. The MASS `cos` and `sin` functions are automatically linked if you export either one of them.

Using the Basic Linear Algebra Subprograms (BLAS)

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

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

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

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

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

"Linking the libxlopt library" on page 64 describes how to link to the XL C/C++ `libxlopt` library if you are also using a third-party BLAS library.

BLAS function syntax

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

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

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

The parameters are as follows:

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

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

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

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

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

alpha
is the scaling constant for matrix *a*

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

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

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

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

beta
is the scaling constant for vector *y*

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

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

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

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

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

The parameters are as follows:

transa
is a single character indicating the form of the input matrix *a*, where:

- `'N'` or `'n'` indicates that *a* is to be used in the computation
- `'T'` or `'t'` indicates that the transpose of *a* is to be used in the computation

transb
is a single character indicating the form of the input matrix *b*, where:

- `'N'` or `'n'` indicates that *b* is to be used in the computation
- `'T'` or `'t'` indicates that the transpose of *b* is to be used in the computation

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

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

m represents:

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

and:

- the number of rows in matrix *b*, if `'N'` or `'n'` is used for the *transb* parameter

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

m must be greater than or equal to zero.

alpha
is the scaling constant for matrix *a*

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

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

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

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

beta
is the scaling constant for matrix *c*

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

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

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

Linking the libxlopt library

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

Chapter 10. Parallelizing your programs

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

- Automatic parallelization of countable program loops, which are defined in "Countable loops." An overview of the compiler's automatic parallelization capabilities is provided in "Enabling automatic parallelization" on page 67.
- Explicit parallelization of C and 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 67.

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

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

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

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

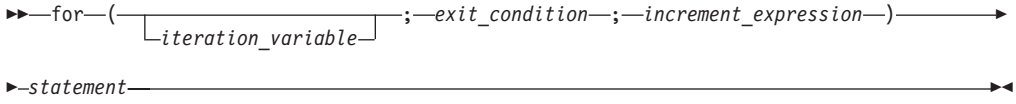
Related information

- "Using shared-memory parallelism (SMP)" on page 41

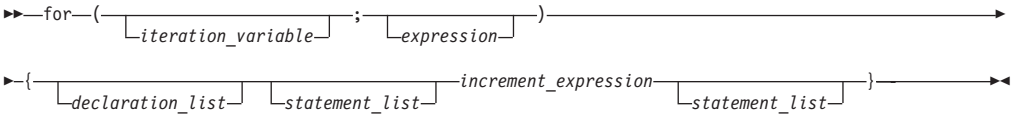
Countable loops

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

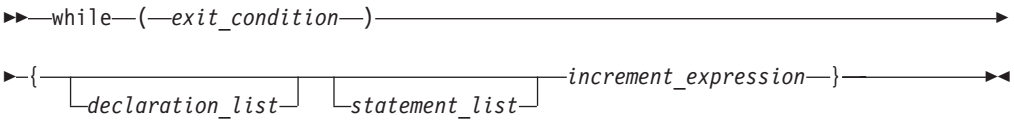
Countable for loop syntax with single statement



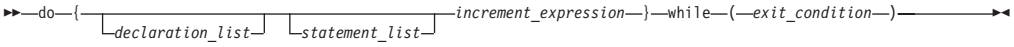
Countable for loop syntax with statement block



Countable while loop syntax



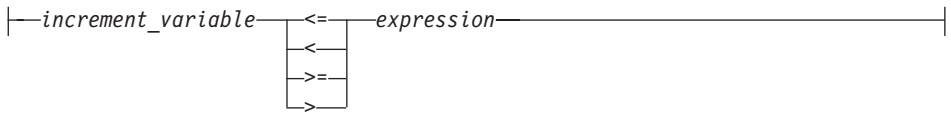
Countable do while loop syntax



The following definitions apply to the above syntax diagrams:

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

exit_condition
takes the following form:



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

increment_expression
takes any of the following forms:

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

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

Enabling automatic parallelization

The compiler can automatically locate and where possible parallelize all countable loops in your program code. A loop is considered to be *countable* if it has any of the forms shown in "Countable loops" on page 65, and:

- There is no branching into or out of the loop.
- The increment expression is not within a critical section.

In general, a countable loop is automatically parallelized only if all of the following conditions are met:

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

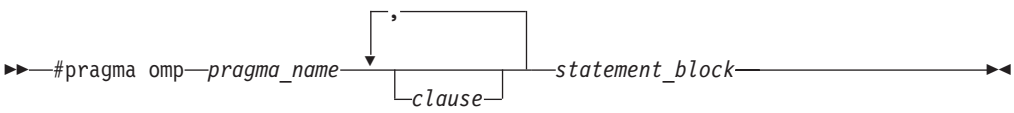
Using OpenMP directives

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

Pragmas fall into four general categories:

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

OpenMP directive syntax



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

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

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

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

For a pragma-by-pragma description of the OpenMP directives, refer to "Pragma directives for parallel processing" in the *XL C/C++ Compiler Reference*.

Shared and private variables in a parallel environment

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

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

- Variables with `static` storage duration are shared.
- Dynamically allocated objects are shared.
- Variables with automatic storage duration are private.
- Variables in heap allocated memory are shared. There can be only one shared heap.
- All variables defined outside a parallel construct become shared when the parallel loop is encountered.
- Loop iteration variables are private within their loops. The value of the iteration variable after the loop is the same as if the loop were run sequentially.
- Memory allocated within a parallel loop by the `alloca` function persists only for the duration of one iteration of that loop, and is private for each thread.

The following code segments show examples of these default rules:

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

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

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

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

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


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


int E2;                         /*shared static */

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

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

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

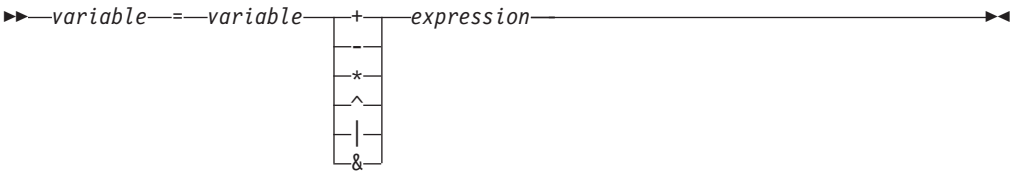
Some OpenMP preprocessor directives let you specify visibility context for selected data variables. A brief summary of data scope attribute clauses are listed below:

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

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

Reduction operations in parallelized loops

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



where:

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

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

expression
is any valid expression.

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

Notices

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

COPYRIGHT LICENSE:

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

Programming interface information

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

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

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

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

Trademarks and service marks

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

- AIX
- IBM
- IBM (logo)
- POWER
- POWER3
- POWER4
- POWER5
- PowerPC
- pSeries

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

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

Industry standards

The following standards are supported:

- The C language is consistent with the International Standard for Information Systems-Programming Language C (ISO/IEC 9899-1999 (E)).
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).
- The C++ language is also consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:2003 (E)).
- The C and C++ languages are consistent with the OpenMP C and C++ Application Programming Interface Version 2.5.

Index

Special characters

Numerics

A

B

C

D

E

F

H

I

L

M

O

P

R

S

T

V

X

Program Number: 5724-M16

SC09-8014-00