

IBM XL C/C++ for Linux, V13.1.2



Optimization and Programming Guide for Little Endian Distributions

Version 13.1.2

IBM XL C/C++ for Linux, V13.1.2



Optimization and Programming Guide for Little Endian Distributions

Version 13.1.2

Note

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

First edition

This edition applies to IBM XL C/C++ for Linux, V13.1.2 (Program 5765-J08; 5725-C73) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

© Copyright IBM Corporation 1996, 2015.

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

Contents

About this document	v
Who should read this document.	v
How to use this document.	v
How this document is organized	v
Conventions	vi
Related information.	ix
IBM XL C/C++ information	ix
Standards and specifications	x
Other IBM information.	xi
Other information	xi
Technical support	xi
How to send your comments	xi

Chapter 1. Porting from 32-bit to 64-bit mode 1

Assigning long values	1
Assigning constant values to long variables	2
Bit-shifting long values	3
Assigning pointers	3
Aligning aggregate data	4
Calling Fortran code.	4

Chapter 2. Using XL C/C++ with Fortran 5

Identifiers	5
Corresponding data types	6
Character and aggregate data.	8
Function calls and parameter passing	8
Pointers to functions.	9
Sample program: C/C++ calling Fortran	9

Chapter 3. Aligning data 11

Using alignment modes	11
Alignment of aggregates	12
Alignment of bit-fields.	13
Using alignment modifiers	14

Chapter 4. Handling floating-point operations 17

Floating-point formats.	17
Handling multiply-add operations.	17
Compiling for strict IEEE conformance	18
Handling floating-point constant folding and rounding	18
Matching compile-time and runtime rounding modes	19
Handling floating-point exceptions	20

Chapter 5. Using C++ constructors . . . 21

Using delegating constructors (C++11)	21
---------------------------------------	----

Chapter 6. The C++ template model . . . 23

Chapter 7. Constructing a library 25

Compiling and linking a library	25
Compiling a static library.	25
Compiling a shared library	25
Linking a library to an application.	25
Linking a shared library to another shared library	26
Initializing static objects in libraries (C++)	26
Assigning priorities to objects	26
Order of object initialization across libraries	27

Chapter 8. Optimizing your applications 31

Distinguishing between optimization and tuning	31
Steps in the optimization process	32
Basic optimization	32
Optimizing at level 0	32
Optimizing at level 2	33
Advanced optimization	34
Optimizing at level 3	35
An intermediate step: adding -qhot suboptions at level 3	36
Optimizing at level 4	36
Optimizing at level 5	37
Tuning for your system architecture	38
Getting the most out of target machine options	38
Using high-order loop analysis and transformations	39
Getting the most out of -qhot	40
Using shared-memory parallelism (SMP)	41
Getting the most out of -qsmpr	42
Using interprocedural analysis	43
Getting the most from -qipa	44
Using profile-directed feedback.	45
Viewing profiling information with showpdf	48
Object level profile-directed feedback.	51
Marking variables as local or imported	53
Getting the most out of -qdatalocal	53
Using compiler reports to diagnose optimization opportunities.	55
Parsing compiler reports with development tools	56
Other optimization options	56

Chapter 9. Debugging optimized code 59

Understanding different results in optimized programs	59
Debugging in the presence of optimization	60

Chapter 10. Coding your application to improve performance 63

Finding faster input/output techniques	63
Reducing function-call overhead	63
Using template explicit instantiation declarations (C++11).	65
Managing memory efficiently (C++ only)	65

Optimizing variables	66
Manipulating strings efficiently.	66
Optimizing expressions and program logic	67
Optimizing operations in 64-bit mode	68
Using rvalue references (C++11)	68
Using visibility attributes (IBM extension)	70
Types of visibility attributes	71
Rules of visibility attributes	73
Propagation rules (C++ only)	78
Specifying visibility attributes using the -fvisibility option	81
Specifying visibility attributes using pragma preprocessor directives	81

Chapter 11. Using the high performance libraries 85

Using the Mathematical Acceleration Subsystem libraries (MASS).	85
Using the scalar library	86
Using the vector libraries.	88
Using the SIMD libraries	92
Compiling and linking a program with MASS.	95
Using the Basic Linear Algebra Subprograms – BLAS	96

BLAS function syntax	97
Linking the libxlopt library	99

Chapter 12. Parallelizing your programs 101

Countable loops	101
Enabling automatic parallelization	103
Data sharing attribute rules.	103
Using OpenMP directives	105
Shared and private variables in a parallel environment.	106
Reduction operations in parallelized loops.	107

Chapter 13. Vector element order toggling. 109

Program migration from big endian systems	112
---	-----

Notices 113

Trademarks	115
----------------------	-----

Index 117

About this document

This guide discusses advanced topics related to the use of IBM® XL C/C++ for Linux, V13.1.2, with a particular focus on program portability and optimization. The guide provides both reference information and practical tips for getting the most out of the compiler's capabilities through recommended programming practices and compilation procedures.

Who should read this document

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

How to use this document

This document uses a task-oriented approach to present the topics by concentrating on a specific programming or compilation problem in each section. Each topic contains extensive cross-references to the relevant sections of the reference guides in the IBM XL C/C++ for Linux, V13.1.2 documentation set, which provides detailed descriptions of compiler options and pragmas, and specific language extensions.

How this document is organized

This guide includes the following chapters:

- Chapter 1, "Porting from 32-bit to 64-bit mode," on page 1 discusses common problems that arise when you port existing 32-bit applications to 64-bit mode, and it provides recommendations for avoiding these problems.
- Chapter 2, "Using XL C/C++ with Fortran," on page 5 discusses considerations for calling Fortran code from XL C/C++ programs.
- Chapter 3, "Aligning data," on page 11 discusses options available for controlling the alignment of data in aggregates, such as structures and classes.
- 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++ constructors," on page 21 discusses delegating constructors that can concentrate common initializations in one constructor.
- Chapter 6, "The C++ template model," on page 23 discusses options for compiling programs that include C++ templates.
- Chapter 7, "Constructing a library," on page 25 discusses how to compile and link static and shared libraries and how to specify the initialization order of static objects in C++ programs.
- Chapter 8, "Optimizing your applications," on page 31 discusses various options provided by the compiler for optimizing your programs, and it provides recommendations for use of these options.

- Chapter 9, “Debugging optimized code,” on page 59 discusses the potential usability problems of optimized programs and the options that can be used to debug the optimized code.
- Chapter 10, “Coding your application to improve performance,” on page 63 discusses recommended programming practices and coding techniques for enhancing program performance and compatibility with the compiler's optimization capabilities.
- Chapter 11, “Using the high performance libraries,” on page 85 discusses two performance 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 12, “Parallelizing your programs,” on page 101 provides an overview of options offered by XL C/C++ for creating multi-threaded programs, including OpenMP language constructs.
- Chapter 13, “Vector element order toggling,” on page 109 discusses that if users want to consistently use the instructions generated by vector built-in functions, users need to make all existing Vector Multimedia Extension (VMX) and Vector Scalar Extension (VSX) load and store built-in functions operate on the vectors in registers in the same vector element order, either little-endian or big-endian element order.

Conventions

Typographical conventions

The following table shows the typographical conventions used in the IBM XL C/C++ for Linux, V13.1.2 information.

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Lowercase commands, executable names, compiler options, and directives.	The compiler provides basic invocation commands, xlc and xlc (xlc++), along with several other compiler invocation commands to support various C/C++ language levels and compilation environments.
<i>italics</i>	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 <i>size</i> parameter if you return more than the <i>size</i> requested.
<u>underlining</u>	The default setting of a parameter of a compiler option or directive.	nomaf <u>maf</u>
monospace	Programming keywords and library functions, compiler builtins, examples of program code, command strings, or user-defined names.	To compile and optimize myprogram.c, enter: xlc myprogram.c -O3.

Qualifying elements (icons)

Most features described in this information apply to both C and C++ languages. In descriptions of language elements where a feature is exclusive to one language, or where functionality differs between languages, this information uses icons to delineate segments of text as follows:

Table 2. Qualifying elements

Qualifier/Icon	Meaning
C only begins   C only ends	The text describes a feature that is supported in the C language only; or describes behavior that is specific to the C language.
C++ only begins   C++ only ends	The text describes a feature that is supported in the C++ language only; or describes behavior that is specific to the C++ language.
IBM extension begins   IBM extension ends	The text describes a feature that is an IBM extension to the standard language specifications.
C11 begins   C11 ends	The text describes a feature that is introduced into standard C as part of C11.
C++11 begins   C++11 ends	The text describes a feature that is introduced into standard C++ as part of C++11.
C++14 begins   C++14 ends	The text describes a feature that is introduced into standard C++ as part of C++14.

Syntax diagrams

Throughout this information, diagrams illustrate XL C/C++ syntax. This section helps you to interpret and use those diagrams.

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

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

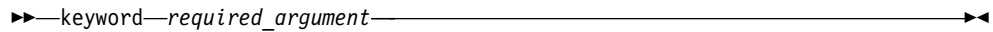
The  symbol indicates that the command, directive, or statement syntax is continued on the next line.

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

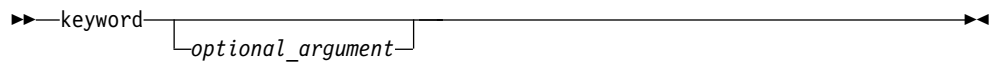
The — $\blacktriangleleft$ symbol indicates the end of a command, directive, or statement.

Fragments, which are diagrams of syntactical units other than complete commands, directives, or statements, start with the $\mid$ — symbol and end with the — $\mid$ symbol.

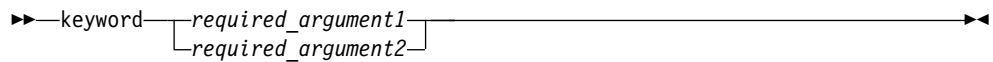
- Required items are shown on the horizontal line (the main path):



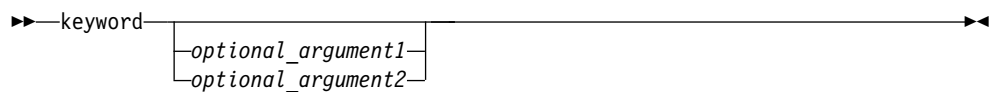
- Optional items are shown below the main path:



- 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.



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



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



- Keywords are shown in nonitalic letters and should be entered exactly as shown.
- Variables are shown in italicized lowercase letters. They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Example of a syntax statement

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

The following list explains the syntax statement:

- Enter the keyword EXAMPLE.

- Enter a value for *char_constant*.
- Enter a value for *a* or *b*, but not for both.
- Optionally, enter a value for *c* or *d*.
- Enter at least one value for *e*. If you enter more than one value, you must put a comma between each.
- Optionally, enter the value of at least one *name* for *name_list*. If you enter more than one value, you must put a comma between each *name*.

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

Examples in this information

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

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

Related information

The following sections provide related information for XL C/C++:

IBM XL C/C++ information

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

- Quick Start Guide
The Quick Start Guide ([quickstart.pdf](#)) is intended to get you started with IBM XL C/C++ for Linux, V13.1.2. It is located by default in the XL C/C++ directory and in the \quickstart directory of the installation DVD.
- README files
README files contain late-breaking information, including changes and corrections to the product information. README files are located by default in the XL C/C++ directory, and in the root directory and subdirectories of the installation DVD.
- Installable man pages
Man pages are provided for the compiler invocations and all command-line utilities provided with the product. Instructions for installing and accessing the man pages are provided in the *IBM XL C/C++ for Linux, V13.1.2 Installation Guide*.
- Online product documentation
The fully searchable HTML-based documentation is viewable in IBM Knowledge Center at http://www.ibm.com/support/knowledgecenter/SSXVZZ_13.1.2/com.ibm.compilers.linux.doc/welcome.html.
- PDF documents
PDF documents are available on the web at <http://www.ibm.com/support/docview.wss?uid=swg27036675>.

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

Table 3. XL C/C++ PDF files

Document title	PDF file name	Description
<i>IBM XL C/C++ for Linux, V13.1.2 Installation Guide, GC27-6540-01</i>	install.pdf	Contains information for installing XL C/C++ and configuring your environment for basic compilation and program execution.
<i>Getting Started with IBM XL C/C++ for Linux, V13.1.2, GI13-2875-01</i>	getstart.pdf	Contains an introduction to the XL C/C++ product, with information about setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
<i>IBM XL C/C++ for Linux, V13.1.2 Compiler Reference, SC27-6570-01</i>	compiler.pdf	Contains information about the various compiler options, pragmas, macros, environment variables, and built-in functions.
<i>IBM XL C/C++ for Linux, V13.1.2 Language Reference, SC27-6550-01</i>	langref.pdf	Contains information about language extensions for portability and conformance to nonproprietary standards.
<i>IBM XL C/C++ for Linux, V13.1.2 Optimization and Programming Guide, SC27-6560-01</i>	proguide.pdf	Contains information about advanced programming topics, such as application porting, interlanguage calls with Fortran code, library development, application optimization, and the XL C/C++ high-performance libraries.

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

More information related to XL C/C++, including IBM Redbooks® publications, white papers, and other articles, is available on the web at <http://www.ibm.com/support/docview.wss?uid=swg27036675>.

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

Standards and specifications

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

- *Information Technology - Programming languages - C, ISO/IEC 9899:1990*, also known as C89.
- *Information Technology - Programming languages - C, ISO/IEC 9899:1999*, also known as C99.
- *Information Technology - Programming languages - C, ISO/IEC 9899:2011*, also known as C11.
- *Information Technology - Programming languages - C++, ISO/IEC 14882:1998*, also known as C++98.
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2003*, also known as *Standard C++*.
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2011*, also known as C++11 (Partial support).
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2014*, also known as C++14 (Partial support).

- *AltiVec Technology Programming Interface Manual*, Motorola Inc. This specification for vector data types, to support vector processing technology, is available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf.
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985.
- *OpenMP Application Program Interface Version 3.1* (full support) and *OpenMP Application Program Interface Version 4.0* (partial support), available at <http://www.openmp.org>

Other IBM information

- *ESSL product documentation* available at http://www.ibm.com/support/knowledgecenter/SSFHY8/essl_welcome.html?lang=en

Other information

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

Technical support

Additional technical support is available from the XL C/C++ Support page at http://www.ibm.com/support/entry/portal/overview/software/rational/xl_c~c++_for_linux. This page provides a portal with search capabilities to a large selection of Technotes and other support information.

If you cannot find what you need, you can send an email to compinfo@cn.ibm.com.

For the latest information about XL C/C++, visit the product information site at <http://www.ibm.com/software/products/us/en/xlcpp-linux/>.

How to send your comments

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

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

Chapter 1. Porting from 32-bit to 64-bit mode

IBM XL C/C++ for Linux, V13.1.2 supports only 64-bit compilation mode, which means you can use the XL C/C++ compiler to develop only 64-bit applications.

You might want to port existing 32-bit applications to the 64-bit IBM XL C/C++ for Linux, V13.1.2. However, this 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 4. Size and alignment of data types in 32-bit and 64-bit modes

Data type	32-bit mode		64-bit mode	
	Size	Alignment	Size	Alignment
long, signed long, unsigned long	4 bytes	4-byte boundaries	8 bytes	8-byte boundaries
pointer	4 bytes	4-byte boundaries	8 bytes	8-byte boundaries
size_t (defined in the header file <stddef>)	4 bytes	4-byte boundaries	8 bytes	8-byte boundaries
ptrdiff_t (defined in the header file <stddef>)	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”
- “Assigning pointers” on page 3
- “Aligning aggregate data” on page 4
- “Calling Fortran code” on page 4

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

Related information in the XL C/C++ Compiler Reference



Compile-time and link-time environment variables

Assigning long values

The limits of long type integers defined in the `limits.h` standard library header file are shown in the following table.

Table 5. Constant limits of long integers in 64-bit mode

Symbolic constant	Value	Hexadecimal	Decimal
LONG_MIN (smallest signed long)	-2^{63}	0x8000000000000000L	-9,223,372,036,854,775,808
LONG_MAX (largest signed long)	$2^{63}-1$	0x7FFFFFFFFFFFFFFFL	+9,223,372,036,854,775,807

Table 5. Constant limits of long integers in 64-bit mode (continued)

Symbolic constant	Value	Hexadecimal	Decimal
ULONG_MAX (largest unsigned long)	$2^{64}-1$	0xFFFFFFFFFFFFFFFFUL	+18,446,744,073,709,551,615

These differences have the following implications:

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

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

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

Assigning constant values to long variables

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

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

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

Table 6. Unexpected boundary results of constants assigned to long types

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

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

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

Bit-shifting long values

The examples in Table 7 show the effects of performing a bit-shift on long constants using the following code segment:

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

Table 7. Results of bit-shifting long values

Initial value	Symbolic constant	Value after bit shift by one bit
0x7FFFFFFFL	INT_MAX	0x00000000FFFFFFFE
0x80000000L	INT_MIN	0x0000000100000000
0xFFFFFFFFL	UINT_MAX	0x00000001FFFFFFFFFE

In 32-bit mode, 0xFFFFFFFF is negative. In 64-bit mode, 0x00000000FFFFFFFFFE and 0x00000001FFFFFFFFFE are both positive.

Assigning pointers

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

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

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

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

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

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

- Prototype any functions that return a pointer, where possible by using the appropriate header file.

- Ensure that the type of parameter you are passing in a function (pointer or int) call matches the type expected by the function being called.
- For applications that treat pointers as an integer type, use type long or unsigned long.

Aligning aggregate data

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

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

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

For detailed information about aligning data structures, including structures that contain bit fields, see Chapter 3, “Aligning data,” on page 11.

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

Table 8. 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” on page 6
- “Character and aggregate data” on page 8
- “Function calls and parameter passing” on page 8
- “Pointers to functions” on page 9

In topic “Sample program: C/C++ calling Fortran” on page 9, an example of a C program that calls a Fortran subroutine is provided.

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

Related information:

“Calling Fortran code” on page 4

Identifiers

C++ functions callable from Fortran should be declared with extern "C" to avoid name mangling. For details, see the appropriate section about options and conventions for mixing Fortran with C/C++ code in the *XL Fortran Optimization and Programming Guide*.

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

- Avoid using uppercase letters in identifiers. Although XL Fortran folds external identifiers to lowercase by default, the Fortran compiler can be set to distinguish external names by case.
- Avoid using long identifier names. The maximum number of significant characters in XL Fortran identifiers is 250¹.

Note:

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

Corresponding data types

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

Table 9. Correspondence of data types between C/C++ and Fortran

C and C++ data types	Fortran data types	
	Types	Types with kind type parameters from the ISO_C_BINDING module
bool (C++) _Bool (C)	LOGICAL*1 or LOGICAL(1)	LOGICAL(C_BOOL)
char	CHARACTER	CHARACTER(C_CHAR)
signed char	INTEGER*1 or INTEGER(1)	INTEGER(C_SIGNED_CHAR)
unsigned char	LOGICAL*1 or LOGICAL(1)	
signed short int	INTEGER*2 or INTEGER(2)	INTEGER(C_SHORT)
unsigned short int	LOGICAL*2 or LOGICAL(2)	
int	INTEGER*4 or INTEGER(4)	INTEGER(C_INT)
unsigned int	LOGICAL*4 or LOGICAL(4)	
signed long int	INTEGER*8 or INTEGER(8)	INTEGER(C_LONG)
unsigned long int	INTEGER*8 or INTEGER(8)	
signed long long int	INTEGER*8 or INTEGER(8)	INTEGER(C_LONG_LONG)
unsigned long long int	LOGICAL*8 or LOGICAL(8)	
size_t	INTEGER*8 or INTEGER(8)	INTEGER(C_SIZE_T)
intptr_t	INTEGER*8 or INTEGER(8)	INTEGER(C_INTPTR_T)
intmax_t	INTEGER*8 or INTEGER(8)	INTEGER(C_INTMAX_T)
int8_t	INTEGER*1 or INTEGER(1)	INTEGER(C_INT8_T)
int16_t	INTEGER*2 or INTEGER(2)	INTEGER(C_INT16_T)
int32_t	INTEGER*4 or INTEGER(4)	INTEGER(C_INT32_T)
int64_t	INTEGER*8 or INTEGER(8)	INTEGER(C_INT64_T)
int_least8_t	INTEGER*1 or INTEGER(1)	INTEGER(C_INT_LEAST8_T)

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

C and C++ data types	Fortran data types	
	Types	Types with kind type parameters from the ISO_C_BINDING module
int_least16_t	INTEGER*2 or INTEGER(2)	INTEGER(C_INT_LEAST16_T)
int_least32_t	INTEGER*4 or INTEGER(4)	INTEGER(C_INT_LEAST32_T)
int_least64_t	INTEGER*8 or INTEGER(8)	INTEGER(C_INT_LEAST64_T)
int_fast8_t	INTEGER, INTEGER*4, or INTEGER(4)	INTEGER(C_INT_FAST8_T)
int_fast16_t	INTEGER*4 or INTEGER(4)	INTEGER(C_INT_FAST16_T)
int_fast32_t	INTEGER*4 or INTEGER(4)	INTEGER(C_INT_FAST32_T)
int_fast64_t	INTEGER*8 or INTEGER(8)	INTEGER(C_INT_FAST64_T)
float	REAL, REAL*4, or REAL(4)	REAL(C_FLOAT)
double	REAL*8, REAL(8), or DOUBLE PRECISION	REAL(C_DOUBLE)
long double	REAL*16 or REAL(16)	REAL(C_LONG_DOUBLE)
float _Complex	COMPLEX*4, COMPLEX(4), COMPLEX*8, or COMPLEX(8)	COMPLEX(C_FLOAT_COMPLEX)
double _Complex	COMPLEX*8, COMPLEX(8), COMPLEX*16, or COMPLEX(16)	COMPLEX(C_DOUBLE_COMPLEX)
long double _Complex	COMPLEX*16, COMPLEX(16), COMPLEX*32, or COMPLEX(32)	COMPLEX(C_LONG_DOUBLE_COMPLEX)
struct or union	derived type	
enum	INTEGER*4 or INTEGER(4)	
char[n]	CHARACTER*n or CHARACTER(n)	
array pointer to type, or type []	Dimensioned variable (transposed)	
pointer to function	Functional parameter	
struct with -fpack-struct (-qalign)	Sequence derived type	

Related information in the XL C/C++ Compiler Reference



-fpack-struct (-qalign)

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.
- An n-element C/C++ array is indexed with $0 \dots n-1$, whereas an n-element Fortran array is typically indexed with $1 \dots n$. In addition, Fortran supports user-specified bounds while C/C++ does not.
- C stores array elements in row-major order (array elements in the same row occupy adjacent memory locations). Fortran stores array elements in ascending storage units in column-major order (array elements in the same column occupy adjacent memory locations). The following table shows how a two-dimensional array declared by `A[3][2]` in C and by `A(3,2)` in Fortran, is stored:

Table 10. Storage of a two-dimensional array

Storage unit	C and C++ element name	Fortran element name
Lowest	<code>A[0][0]</code>	<code>A(1,1)</code>
	<code>A[0][1]</code>	<code>A(2,1)</code>
	<code>A[1][0]</code>	<code>A(3,1)</code>
	<code>A[1][1]</code>	<code>A(1,2)</code>
	<code>A[2][0]</code>	<code>A(2,2)</code>
Highest	<code>A[2][1]</code>	<code>A(3,2)</code>

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

Function calls and parameter passing

Functions must be prototyped equivalently in both C/C++ and Fortran.

In C and 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.

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

Pointers to functions

A function pointer is a data type whose value is a function address. In Fortran, a dummy argument that appears in an EXTERNAL statement is a function pointer. Starting from the Fortran 2003 standard, Fortran variables of type C_FUNPTR are interoperable with function pointers. Function pointers are supported in contexts such as the target of a call statement or an actual argument of such a statement.

Sample program: C/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. The example includes the following source files:

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

Main program source file: `example.c`

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

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

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

    x = 3;
    y = 3;

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

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

Fortran add function source file: `add.f`

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

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

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

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

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

Execute binary:

```
./add
```

The output is as follows:
The sum of 3 and 7 is 10

Chapter 3. Aligning data

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

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

Using alignment modifiers, you can set the alignment for specific variables or data types within a compilation unit by specifying the exact number of bytes that should be used for the alignment.

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

“Using alignment modifiers” on page 14 discusses the different specifiers, pragmas, and attributes you can use in your source code to override the alignment mode currently in effect, for specific variable declarations. It also provides the rules that govern the precedence of alignment modes and modifiers during compilation.

Related information in the XL C/C++ Compiler Reference



-maltivec

Related external information



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

Using alignment modes

Each data type that is supported by XL C/C++ is aligned along byte boundaries according to platform-specific default alignment modes. The default alignment mode is **linuxppc**.

Each of the valid alignment modes is defined in Table 11, which provides the alignment value, in bytes, for scalar variables of all data types.

Table 11. Alignment settings (values given in bytes)

Data type	Storage	Alignment setting	
		linuxppc	bit_packed
_Bool (C), bool (C++)	1	1	1

Table 11. Alignment settings (values given in bytes) (continued)

Data type	Storage	Alignment setting	
		linuxppc	bit_packed
char, signed char, unsigned char	1	1	1
wchar_t	4	4	1
int, unsigned int	4	4	1
short int, unsigned short int	2	2	1
long int, unsigned long int	8	8	1
long long	8	8	1
float	4	4	1
double	8	8	1
long double	16	16	1
pointer	8	8	1
vector types	16	16	1

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

Notes:

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

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

Related information in the XL C/C++ Compiler Reference



`-fpack-struct (-qalign)`

Alignment of aggregates

The data contained in Table 11 on page 11 (in “Using alignment modes” on page 11) apply to scalar variables, and variables that are members of aggregates such as structures, unions, and classes. The following rules apply to aggregate variables, namely structures, unions or classes, as a whole (in the absence of any modifiers):

- For all alignment modes, the size of an aggregate is the smallest multiple of its alignment value that can encompass all of the members of the aggregate.
-  **C** Empty aggregates are assigned a size of 0 bytes. As a result, two distinct variables might have the same address.
-  **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.

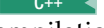
- For all alignment modes, the alignment of an aggregate is equal to the largest alignment value of any of its members. With the exception of packed alignment modes, members whose natural alignment is smaller than that of their aggregate's alignment are padded with empty bytes.
- Aligned aggregates can 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.

Notes:

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

For rules on the alignment of aggregates containing bit fields, see “Alignment of bit-fields.”

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.

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

 The length of a bit-field can exceed the length of its base type, but the remaining bits are used to pad the field, and do 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, and an `int` bit-field uses 31 bits. 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.

Using alignment modifiers

XL C/C++ also provides alignment modifiers, with which you can exercise even finer-grained control over alignment, at the level of declaration or definition of individual variables or aggregate members. Available modifiers are as follows:

#pragma pack(...)

Valid application:

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

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

Valid values:

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

__attribute__((aligned(n)))

Valid application:

As a variable attribute, it applies to a single aggregate (as a whole), namely a structure, union, or class; or to an individual member of an aggregate.¹

As a type attribute, it applies to all aggregates declared of that type. If it is applied to a typedef declaration, it applies to all instances of that type.²

Effect:

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

Valid values:

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

__attribute__((packed))

Valid application:

As a variable attribute, it applies to simple variables or individual members of an aggregate, namely a structure or class¹. 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.

Notes:

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

Related information in the *XL C/C++ Compiler Reference*



#pragma pack

Related information in the *XL C/C++ Language Reference*



The aligned type attribute (IBM extension)



The packed type attribute (IBM extension)



Type attributes (IBM extension)



The aligned variable attribute (IBM extension)



The packed variable attribute (IBM extension)

Chapter 4. Handling floating-point operations

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

- “Floating-point formats”
- “Handling multiply-add operations”
- “Compiling for strict IEEE conformance” on page 18
- “Handling floating-point constant folding and rounding” on page 18
- “Handling floating-point exceptions” on page 20

Floating-point formats

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

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

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

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

Handling multiply-add operations

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

Related information in the XL C/C++ Compiler Reference



-qfloat

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, use the following options:

- Use the **-qfloat=nomaf** compiler option.
- If the program changes the rounding mode at run time, use the **-qfloat=rrm** option.
- If the data or program code contains signaling NaN values (NaNs), use any of the following groups of options. (A signaling NaN is different from a quiet NaN; you must explicitly code it into the program or data, or create it by using the **-qinitauto** compiler option.)
 - The **-qfloat=nans** and **-qstrict=nans** options
 - The **-qfloat=nans** and **-qstrict** options
- If you compile with **-O3**, **-O4**, or **-O5**, include the option **-qstrict** after it. You can also use the suboptions of **-qstrict** to refine the level of control for the transformations performed by the optimizers.

Related information:

“Advanced optimization” on page 34

Related information in the XL C/C++ Compiler Reference



-qfloat



-qstrict



-qinitauto

Handling floating-point constant folding and rounding

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

- The compile-time rounding mode is different from the execution-time rounding mode. By default, both are round-to-nearest; however, if your program changes the execution-time rounding mode, to avoid differing results, perform either of the following operations:
 - Change the compile-time rounding mode to match the execution-time mode, by compiling with the appropriate **-y** option. For more information and an example, see “Matching compile-time and runtime rounding modes” on page 19.
 - Suppress folding by compiling with the **-qfloat=nofold** option.
- Expressions like $a+b*c$ are partially or fully evaluated at compile time. The results might be different from those produced at execution time, because $b*c$ might be rounded before being added to a , while the runtime multiply-add instruction does not use any intermediate rounding. To avoid differing results, perform either of the following operations:

- Suppress the use of multiply-add instructions by compiling with the **-qfloat=nomaf** option.
- Suppress folding by compiling with the **-qfloat=nofold** option.
- An operation produces an infinite, NaN, or underflow to zero result. Compile-time folding prevents execution-time detection of an exception, even if you compile with the **-ftrapping-math (-qflttrap)** option. To avoid missing these exceptions, suppress folding with the **-qfloat=nofold** option.

Related information:

“Handling floating-point exceptions” on page 20

Related information in the XL C/C++ Compiler Reference



-qfloat



-qstrict



-ftrapping-math (-qflttrap)

Matching compile-time and runtime rounding modes

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

```
#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 = 0.3333333333333331
1/3 with execution-time rounding to zero = 0.3333333333333331
1/3 with execution-time rounding to nearest = 0.3333333333333331
1/3 with execution-time rounding to +infinity = 0.3333333333333337
1/3 with execution-time rounding to -infinity = 0.3333333333333331

```

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 = 0.3333333333333337

```

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

Related information in the XL C/C++ Compiler Reference



-qfloat



-y

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 **-ftrapping-math (-qflttrap)** option or call C or operating system functions to detect these types of exceptions. If you enable floating-point traps without using the **-ftrapping-math (-qflttrap)** option, use the **-qfloat=fenv** option.

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

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

Related information in the XL C/C++ Compiler Reference



-qfloat



-ftrapping-math (-qflttrap)

Chapter 5. Using C++ constructors

> C++11

Before C++11, common initialization in multiple constructors of the same class cannot be concentrated in one place in a robust and maintainable manner. Starting from C++11, you can use the following basic approach to solve this problem:

Using delegating constructors:

With the delegating constructors feature, you can concentrate common initialization in one constructor, which can make the program more readable and maintainable. Delegating constructors help reduce code size and collective size of object files.

C++11 <

Related information:

“Using delegating constructors (C++11)”

Related information in the *XL C/C++ Compiler Reference*



-qlanglvl

Using delegating constructors (C++11)

Note: IBM supports the majority of C++11 features, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard.

Syntactically, *delegating constructors* and *target constructors* present the same interface as other constructors.

Consider the following points when you use the delegating constructors feature:

- Call the target constructor implementation in such a way that virtual bases, direct nonvirtual bases, and class members are initialized by the target constructor as appropriate.
- The feature has minimal impact on compile-time and runtime performance. However, use of default arguments with an existing constructor is recommended in place of a delegating constructor where possible. Without inlining and interprocedural analysis, runtime performance might degrade because of function call overhead and increased opacity.

Chapter 6. The C++ template model

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

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

Each compiler implements templates according to a model that determines the meaning of a template at various stages of the translation of a program. In particular, the compiler determines what the various constructs in a template mean when the template is instantiated. Name lookup is an essential ingredient of the compilation model.

Template instantiation is a process that generates types and functions from generic template definitions. The concept of instantiation of C++ templates is fundamental but also intricate because the definitions of entities generated by a template are no longer limited to a single location in the source code. The location of the template, the location where the template is used, and the locations where the template arguments are defined all contribute to the meaning of the entity.

XL C/C++ supports *Greedy instantiation*. The compiler generates a template instantiation in each compilation unit that uses it. The linker discards the duplicates.

Related information in the XL C/C++ Compiler Reference



-qtmplinst (C++ only)

Chapter 7. Constructing a library

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

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

“Initializing static objects in libraries (C++)” on page 26 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

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

Related information:

Dynamic and static linking

Compiling a static library

To compile a static library, follow this procedure:

1. Compile each source file to get an object file. For example:

```
xlc -c test.c example.c
```
2. Use the `ar` command to add the generated object files to an archive library file. For example:

```
ar -rv libex.a test.o example.o
```

Compiling a shared library

To compile a shared library, follow this procedure:

1. Compile your source files to get an object file. Note that in the case of compiling a shared library, the **-fPIC (-qpvc)** compiler option is also used. For example:

```
xlc -fPIC -c foo.c
```
2. Use the **-shared (-qmkshrobj)** compiler option to create a shared object from the generated object files. For example:

```
xlc -shared -o libfoo.so foo.o
```

Related information in the *XL C/C++ Compiler Reference*



-fPIC (-qpvc)



-shared (-qmkshrobj)

Linking a library to an application

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

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

At compile time, you instruct the linker to search for `libtest.so` in the first directory specified by the `-L` option. If `libtest.so` is not found, the linker searches for `libtest.a`. If neither file is found, the search continues with the next directory specified by the `-L` option.

At run time, the runtime linker searches for `libtest.so` in the first directory specified by the `-R` option. If `libtest.so` is not found, the search continues with the next directory specified by the `-R` option. The path specified by the `-R` option can be overridden at run time by the `LD_LIBRARY_PATH` environment variable.

For additional linkage options, including options that modify the default behavior, see the operating system **ld** documentation .

Related information in the XL C/C++ Compiler Reference

 `-l`

 `-L`

 `-R`

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 -shared -o mylib.so myfile.o -ldirectory -Rdirectory -ltest
```

Related information in the XL C/C++ Compiler Reference

 `-shared (-qmkshrobj)`

 `-R`

 `-L`

Initializing static objects in libraries (C++)

The C++ language definition specifies that all non-local objects with constructors from all the files included in the program must be properly constructed before the main function in a C++ program is executed. 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 27.

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 different platforms, the levels at which you can assign priorities vary among different platforms as follows:

Set the priority level for an entire file

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

Set the priority level for individual objects

To use this approach, use `init_priority` variable attributes in the source files. The `init_priority` attribute 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.

Using priority numbers

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” provides detailed information on the order of initialization of objects.

Related information in the *XL C/C++ Compiler Reference*



`-qpriority`



`-shared (-qmkshrobj)`

Related information in the *XL C/C++ Language Reference*



The `init_priority` variable attribute

Order of object initialization across libraries

Each static library and shared library is loaded and initialized at runtime 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 26. 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. You must use the **-fPIC (-qpic)** option to compile what is to be included in a shared library.

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

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

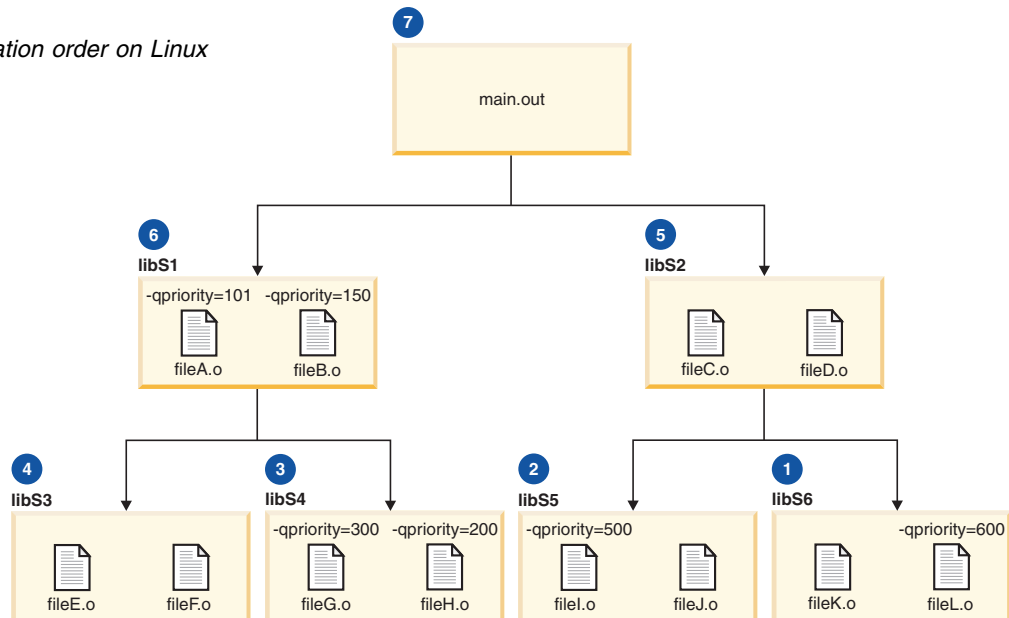
```
xlc -shared -o libS1.so fileA.o fileB.o -L. -R. -lS3 -lS4
xlc -shared -o libS2.so fileC.o fileD.o -L. -R. -lS5 -lS6
```

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).
18	fileB	150	The objects in fileB are initialized next (next priority number in this module).

Sequence	Object	Priority value	Comment
19	main.out	n/a	Initialized last. The objects in main.out are initialized according to their priority.

Related information in the *XL C/C++ Compiler Reference*



-shared (-qmkshrobj)



-W

Chapter 8. Optimizing your applications

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

Distinguishing between optimization and tuning

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

Optimization

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

- Reduce the number of instructions that your application executes to perform critical operations.
- Restructure your object code to make optimal use of the PowerPC architecture.
- Improve memory subsystem usage.

Each basic optimization technique can result in a performance benefit, although not all optimizations can benefit all applications. Consult the “Steps in the optimization process” on page 32 for an overview of the common sequence of steps that you can use to increase the performance of your application.

Tuning

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

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

For instructions, see “Tuning for your system architecture” on page 38.

Steps in the optimization process

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

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

1. The Basic optimization step begins your optimization processes at levels 0 and 2.
2. The Advanced optimization step exposes your application to more intense optimizations at levels 3, 4, and 5.
3. The Using high-order loop analysis and transformations step can help you limit loop execution time.
4. The Using interprocedural analysis step can optimize your entire application at once.
5. The Using profile-directed feedback step focuses optimizations on specific characteristics of your application.
6. The Debugging optimized code step can help you identify issues and problems that can occur with optimized code.

Basic optimization

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

Ensure that your application compiles and executes properly at low optimization levels before trying more aggressive optimizations. This topic discusses two optimizations levels, listed with complementary options in Table 12. The table also includes a column for compiler options that can have a performance benefit at that optimization level for some applications.

Table 12. Basic optimizations

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

Optimizing at level 0

Benefits at level 0

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

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

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

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

For more information about tuning, see “Tuning for your system architecture” on page 38.

Related information in the XL C/C++ Compiler Reference

 **-mcpu**

Optimizing at level 2

Benefits at level 2

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

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

In C, compile with **-qlibansi** unless your application defines functions with names identical to those of library functions. If you encounter problems with **-O2**, consider using **-qalias=noansi** rather than turning off optimization.

Also, ensure that pointers in your C code follow these type restrictions:

- Generic pointers can be `char*` or `void*`.
- Mark all shared variables and pointers to shared variables `volatile`.

Starting to tune at O2

Choosing the right hardware architecture target or family of targets becomes even more important at **-O2** and higher. By targeting the proper hardware, the optimizer can make the best use of the hardware facilities available. If you choose a family of hardware targets, the **-mtune** option can direct the compiler to emit code that is consistent with the architecture choice and that can execute optimally on the

chosen tuning hardware target. With this option, you can compile for a general set of targets and have the code run best on a particular target.

For details on the **-mcpu** and **-mtune** options, see “Tuning for your system architecture” on page 38.

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

- Common subexpression elimination: Eliminates redundant instructions.
- Constant propagation: Evaluates constant expressions at compile time.
- Dead code elimination: Eliminates instructions that a particular control flow does not reach, or that generate an unused result.
- Dead store elimination: Eliminates unnecessary variable assignments.
- Global register allocation: Globally assigns user variables to registers.
- Value numbering: Simplifies algebraic expressions, by eliminating redundant computations.
- Instruction scheduling for the target machine.
- Loop unrolling and software pipelining.
- Moving loop-invariant code out of loops.
- Simplifying control flow.
- Strength reduction and effective use of addressing modes.
- Widening, which merges adjacent load/stores and other operations.
- Pointer aliasing improvements to enhance other optimizations.

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

Advanced optimization

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

After applying “Basic optimization” on page 32 and successfully compiling and executing your application, you can apply more powerful optimization tools. The XL compiler optimization portfolio includes many options for directing advanced optimization, and the transformations that your application undergoes are largely under your control. The discussion of each optimization level in Table 13 includes information on the performance benefits and the possible trade-offs and information on how you can help guide the optimizer to find the best solutions for your application.

Table 13. Advanced optimizations

Optimization Level	Additional options implied	Complementary options	Options with possible benefits
-O3	-qnostrict -qmaxmem=-1 -qhot=level=0	-mcpu -mtune	-qpdf

Table 13. Advanced optimizations (continued)

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

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

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

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

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

Optimizing at level 3

Benefits at level 3

- In-depth memory access analysis
- Better loop scheduling
- High-order loop analysis and transformations (**-qhot=level=0**)
- Inlining of small procedures within a compilation unit by default
- Eliminating implicit compile-time memory usage limits

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

Potential trade-offs at level 3

With the in-depth analysis of **-O3** comes a trade-off in terms of compilation time and memory resources. Also, because **-O3** implies **-qnostrict**, the optimizer can

alter certain floating-point semantics in your application to gain execution speed. This typically involves precision trade-offs as follows:

- Reordering of floating-point computations
- Reordering or elimination of possible exceptions, such as division by zero or overflow
- Using alternative calculations that might give slightly less precise results or not handle infinities or NaNs in the same way

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

Without **-qstrict**, the difference in computation for any one source-level operation is very small in comparison to “Basic optimization” on page 32. Although a small difference can be compounded if the operation is in a loop structure where the difference becomes additive, most applications are not sensitive to the changes that can occur in floating-point semantics.

For information on the **-O** level syntax, see “-O -qoptimize” in the *XL C/C++ Compiler Reference*.

An intermediate step: adding -qhot suboptions at level 3

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

For more information about **-qhot**, see “Using high-order loop analysis and transformations” on page 39.

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

Optimizing at level 4

Benefits at level 4

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

Optimizing at **-04** builds on **-03** by triggering **-qipa=level=1**, which performs interprocedural analysis (IPA), optimizing your entire application as a unit. This option is particularly pertinent to applications that contain a large number of frequently used routines.

To make full use of IPA optimizations, you must specify **-04** on the compilation and link steps of your application build as interprocedural analysis occurs in stages at both compile time and link time.

Beyond **-qipa**, **-04** enables other optimization options:

- **-qhot**

Enables more aggressive HOT transformations to optimize loop constructs and array language.

- **-qarch=auto** and **-qtune=auto**

Optimizes your application to execute on a hardware architecture identical to your build machine. If the architecture of your build machine is incompatible with the execution environment of your application, you must specify a different **-qarch** suboption after the **-04** option. This overrides **-qtune=auto**.

- **-qcache=auto**

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

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

Potential trade-offs at level 4

In addition to the trade-offs already mentioned for **-03**, specifying **-qipa** can significantly increase compilation time, especially at the link step.

The IPA process

1. At compile time optimizations occur on a file-by-file basis, as well as preparation for the link stage. IPA writes analysis information directly into the object files the compiler produces.
2. At the link stage, IPA reads the information from the object files and analyzes the entire application.
3. This analysis guides the optimizer on how to rewrite and restructure your application and apply appropriate **-03** level optimizations.

The “Using interprocedural analysis” on page 43 section contains more information on IPA including details on IPA suboptions.

Optimizing at level 5

Benefits at level 5

- Makes most aggressive optimizations available
- Makes full use of loop optimizations and interprocedural analysis

As the highest optimization level, **-05** includes all **-04** optimizations and deepens whole program analysis by increasing the **-qipa** level to 2. Compiling with **-05** also increases how aggressively the optimizer pursues aliasing improvements.

Additionally, if your application contains a mix of C/C++ and Fortran code that you compile using the XL compilers, you can increase performance by compiling and linking your code with the **-O5** option.

Potential trade-offs at level 5

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

Tuning for your system architecture

You can instruct the compiler to generate code for optimal execution on a given microprocessor or architecture family. By selecting appropriate target machine options, you can optimize to suit the broadest possible selection of target processors, a range of processors within a given family of processor architectures or a specific processor.

The following table lists the optimization options that affect individual aspects of the target machine. Using a predefined optimization level sets default values for these individual options.

Table 14. Target machine options

Option	Behavior
-mcpu	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. See “Getting the most out of target machine options” for more information about this option.
-mtune	Biases optimization toward execution on a given microprocessor, without implying anything about the instruction set architecture to be used as a target. See “Getting the most out of target machine options” for more information about this option.
-qcache	Defines a specific cache or memory geometry. The defaults are determined through the setting of -mtune . See “Getting the most out of target machine options” for more information about this option.

Related information in the XL C/C++ Compiler Reference



-mcpu



-qipa



-qcache

Getting the most out of target machine options

Using **-mcpu** or **-qarch** options

Use the **-mcpu** or **-qarch** compiler option to generate instructions that are optimized for a specific machine architecture. For example, if you want to generate an object code that contains instructions optimized for POWER8®, you use **-mcpu=pwr8**. If your application runs on the same machine on which you are compiling it, you can use the **-qarch=auto** option, which automatically detects the specific architecture of the compiling machine and generates code to take advantage of instructions available only on that machine (or on a system that

supports the equivalent processor architecture). Otherwise, use the **-mcpu** or **-qarch** option to specify the smallest possible family of the machines that can run your code reasonably well.

Using **-mtune** or **-qtune** options

Use the **-mtune** or **-qtune** compiler option to control the scheduling of instructions that are optimized for your machine architecture. If you specify a particular architecture with **-mcpu** or **-qarch**, **-mtune** or **-qtune** automatically selects the suboption that generates instruction sequences with the best performance for that architecture.

If you need to create a single binary file that runs on a range of PowerPC hardware, you can use the **-qtune=balanced** option. With this option in effect, optimization decisions made by the compiler are not targeted to a specific version of hardware. Instead, tuning decisions try to include features that are generally helpful across a broad range of hardware and avoid those optimizations that might be harmful on some hardware.

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

The difference between **-qtune=balanced** and other **-qtune** suboptions including **-qtune=auto** is as follows:

- With the **-qtune=balanced** option, the compiler generates instructions that perform reasonably well across a range of Power[®] hardware.
- With other suboptions, the compiler generates instructions that are optimized for that specified versions of hardware architecture and might not perform well on others.

Using **-qcache** options

If you decide to specify your own **-qcache** suboptions, use **-qhot** or **-qsmp** along with it.

Related information in the *XL C/C++ Compiler Reference*



-qhot



-qcache



-mcpu



-mtune

Using high-order loop analysis and transformations

High-order transformations are optimizations that specifically improve the performance of loops through techniques such as interchange, fusion, and unrolling.

The goals of these loop optimizations include:

- Reducing the costs of memory access through the effective use of caches and address translation look-aside buffers
- Overlapping computation and memory access through effective utilization of the data prefetching capabilities provided by the hardware

- Improving the utilization of microprocessor resources through reordering and balancing the usage of instructions with complementary resource requirements
- Generating SIMD vector instructions
- Generating calls to vector math library functions

To enable high-order loop analysis and transformations, use the **-qhot** option, which implies an optimization level of **-02**. The following table lists the suboptions available for **-qhot**.

Table 15. *-qhot suboptions*

Suboption	Behavior
level=0	Instructs the compiler to perform a subset of high-order transformations that enhance performance by improving data locality. This suboption implies -qhot=novector and -qhot=noarraypad . This level is automatically enabled if you compile with -03 .
level=1	This is the default suboption if you specify -qhot with no suboptions. This level is also automatically enabled if you compile with -04 or -05 . This is equivalent to specifying -qhot=vector .
level=2	When used with -qsmp , instructs the compiler to perform the transformations of -qhot=level=1 plus some additional transformation on nested loops. The resulting loop analysis and transformations can lead to more cache reuse and loop parallelization.
vector	When specified with -qnostrict and -qignerrno , or -03 or a higher optimization level, instructs the compiler to transform some loops to use the optimized versions of various math functions contained in the MASS libraries, rather than use the system versions. The optimized versions make different trade-offs with respect to accuracy and exception-handling versus performance. This suboption is enabled by default if you specify -qhot with no suboptions. Also, specifying -qhot=vector with -03 implies -qhot=level=1 .
arraypad	Instructs the compiler to pad any arrays where it infers there might be a benefit and to pad by whatever amount it chooses.

Related information in the XL C/C++ Compiler Reference



-qhot



-qstrict



-qignerrno



-mcpu



-qsimd

Getting the most out of -qhot

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

- Try using **-qhot** along with **-03** for all of your code. It is designed to have a neutral effect when no opportunities for transformation exist. However, it increases compilation time and might have little benefit if the program has no loop processing vectors or arrays. In this case, using **-03 -qnohot** might be better.
- If the runtime performance of your code can significantly benefit from automatic inlining and memory locality optimizations, try using **-04** with **-qhot=level=0** or **-qhot=novector**.

- If you encounter unacceptably long compilation time (this can happen with complex loop nests), try **-qhot=level=0** or **-qnohot**.
- If your code size is unacceptably large, try reducing the inlining level or using **-qcompact** along with **-qhot**.
- You can compile some source files with the **-qhot** option and some files without the **-qhot** option, allowing the compiler to improve only the parts of your code that need optimization.
- Use **-qreport** along with **-qsimd=auto** to generate a loop transformation listing. The listing file identifies how loops are transformed in a section marked LOOP TRANSFORMATION SECTION. Use the listing information as feedback about how the loops in your program are being transformed. Based on this information, you might want to adjust your code so that the compiler can transform loops more effectively. For example, you can use this section of the listing to identify non-stride-one references that might prevent loop vectorization.
- Use **-qreport** along with **-qhot** or any optimization option that implies **-qhot** to generate information about nested loops in the LOOP TRANSFORMATION SECTION of the listing file. In addition, when you use **-qprefetch=assistthread** to generate prefetching assist threads, a message Assist thread for data prefetching was generated is also displayed in this section of the report. To generate a list of aggressive loop transformations and parallelizations performed on loop nests in the LOOP TRANSFORMATION SECTION of the listing file, use **-qhot=level=2** and **-qsmp** together with **-qreport**.
- Use **-qassert=refalign**, where appropriate, to assert to the compiler that all pointers inside the compilation unit only point to data that is naturally aligned with respect to the length of the pointer types. With this assertion, the compiler might generate more efficient code. This assertion is particularly useful when you target a SIMD architecture with **-qhot=level=0** or **-qhot=level=1** with the **-qsimd=auto** option.

Related information in the XL C/C++ Compiler Reference

-  **-qcompact**
-  **-qhot**
-  **-qsimd**
-  **-qprefetch**
-  **-qstrict**

Using shared-memory parallelism (SMP)

Most IBM pSeries machines are capable of shared-memory parallel processing. You can compile with **-qsmp** to generate the threaded code needed to exploit this capability. The **-qsmp** option implies the **-qhot** option and an optimization level of **-O2** or higher.

The following table lists the most commonly used suboptions. Descriptions and syntax of all the suboptions are provided in **-qsmp** in the *XL C/C++ Compiler Reference*. An overview of automatic parallelization, as well as of OpenMP directives is provided in Chapter 12, “Parallelizing your programs,” on page 101.

Table 16. 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 incompatible with -qsmp=auto .
opt	Instructs the compiler to optimize as well as parallelize. The optimization is equivalent to -O2 -qhot in the absence of other optimization options.
noopt	All optimization is turned off. During development, it can be useful to turn off optimization to facilitate debugging.
<i>fine_tuning</i>	Other values for the suboption provide control over thread scheduling, locking, and so on.

Related information in the *XL C/C++ Compiler Reference*



-O, **-qoptimize**



-qsmp



-qhot

Getting the most out of **-qsmp**

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

- Before using **-qsmp** with automatic parallelization, test your programs using optimization and **-qhot** in a single-threaded manner.
- If you are compiling an OpenMP program and do not want automatic parallelization, use **-qsmp=omp:noauto**.
- By default, the runtime environment uses all available processors. Do not set the `XLSMPOPTS=PARTHDS` or `OMP_NUM_THREADS` environment variables unless you want to use fewer than the number of available processors. You might want to set the number of executing threads to a small number or to 1 to ease debugging.
- If you are using a dedicated machine or node, consider setting the `SPINS` and `YIELDS` environment variables (suboptions of the `XLSMPOPTS` environment variable) to 0. Doing so prevents the operating system from intervening in the scheduling of threads across synchronization boundaries such as barriers.
- When debugging an OpenMP program, try using **-qsmp=noopt** (without **-O**) to make the debugging information produced by the compiler more precise.

Related information in the *XL C/C++ Compiler Reference*



-qsmp



-qhot



Invoking the compiler



`XLSMPOPTS`



Environment variables for parallel processing

Using interprocedural analysis

Interprocedural analysis (IPA) enables the compiler to optimize across different files (whole-program analysis), and it can result in significant performance improvements.

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

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

The steps to use IPA are as follows:

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

Table 17. 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.

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

Suboption	Behavior
<code>level=2</code>	Global alias analysis, specialization, interprocedural data flow: • Whole-program alias analysis. This level includes the disambiguation of pointer dereferences and indirect function calls, and the refinement of information about the side effects of a function call. • Intensive intraprocedural optimizations. This can take the form of value numbering, code propagation and simplification, moving code into conditions or out of loops, and elimination of redundancy. • Interprocedural constant propagation, dead code elimination, pointer analysis, code motion across functions, and interprocedural strength reduction. • Procedure specialization (cloning). • Whole program data reorganization.
<code>inline=suboptions</code>	Provides precise control over function inlining.
<code><i>fine_tuning</i></code>	Other values for -qipa provide the ability to specify the behavior of library code, tune program partitioning, read commands from a file, and so on.

Related information in the *XL C/C++ Compiler Reference*



-qipa

Getting the most from **-qipa**

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

- Specify the **-qipa** option on both the compile and the link steps of the entire application. Although you can also use **-qipa** with libraries, shared objects, and executable files, be sure to use **-qipa** to compile the main and exported functions.
- When compiling and linking separately, use **-qipa=noobject** on the compile step for faster compilation.
- When specifying optimization options in a makefile, use the compiler driver (**xlC**) to link with all the compiler options on the link step included.
- As IPA can generate significantly larger object files than traditional compilations, ensure that there is enough space in the `/tmp` directory (at least 200 MB). You can use the `TMPDIR` environment variable to specify a directory with sufficient free space.
- Try varying the **level** suboption if link time is too long. Compiling with **-qipa=level=0** can still be very beneficial for little additional link time.
- Use **-qipa=list=long** to generate a report of functions that were previously inlined. If too few functions are inlined, consider using the **-finline-functions** option.
- To generate data reorganization information in the listing file, specify the optimization level **-qipa=level=2** or **-O5** together with **-qreport**. During the IPA link pass, the data reorganization messages for program variable data will be produced to the data reorganization section of the listing file with the label `DATA REORGANIZATION SECTION`. Reorganizations include array splitting, array transposing, memory allocation merging, array interleaving, and array coalescing.

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

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

Related information in the XL C/C++ Compiler Reference

 `-finline-functions`

 `-qlist`

 `-qipa`

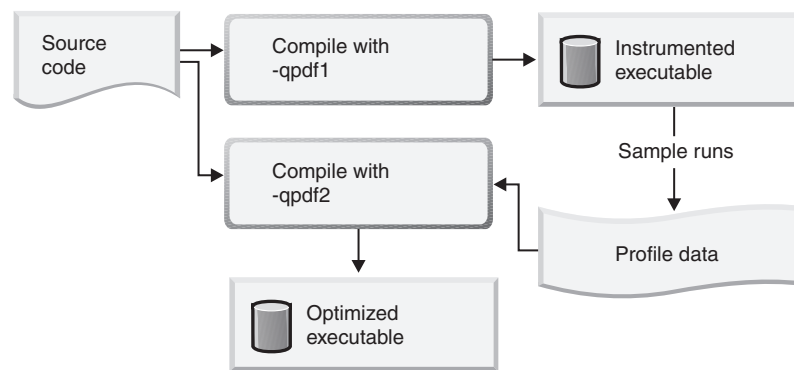
Using profile-directed feedback

You can use profile-directed feedback (PDF) to tune the performance of your application for a typical usage scenario. The compiler optimizes the application based on an analysis of how often branches are taken and blocks of code are run.

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

The following diagram illustrates the PDF process.

Figure 2. Profile-directed feedback



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

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

Notes:

- A PDF map file is generated at this step. It is used for the **showpdf** utility to display part of the profiling information in text or XML format. For details, see “Viewing profiling information with showpdf” on page 48. If you do not need to view the profiling information, specify the **-qnoshowpdf** option at this step so that the PDF map file is not generated. For details of **-qnoshowpdf**, see **-qshowpdf** in the *XL C/C++ Compiler Reference*.
- Although you can specify PDF optimization (**-qpdf**) as early in the optimization level as **-O2**, PDF optimization is recommended at **-O4** and higher.
- You do not have to compile all of the files of the programs with the **-qpdf1** option. In a large application, you can concentrate on those areas of the code that can benefit most from the optimization.
- When option **-O4**, **-O5**, or any level of option **-qipa** is in effect, and you specify the **-qpdf1** option at the link step but not at the compile step, the compiler issues a warning message. The message indicates that you must recompile your program to get all the profiling information.

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

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

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

Notes:

- When you compile your program with the **-qpdf1** or **-qpdf2** option, by default, the **-qipa** option is also invoked with **level=0**.
- To avoid wasting compile and run time, make sure that the `PDFDIR` environment variable is set to an absolute path. Otherwise, you might run the application from a wrong directory, and the compiler cannot locate the profiling information files. When it happens, the program might not be optimized correctly or might be stopped by a segmentation fault. A segmentation fault might also happen if you change the value of the `PDFDIR` environment variable and run the application before the PDF process finishes.

- Avoid using a typical data that can distort the analysis of infrequently executed code paths.
3. If you have several PDF files, use the **mergepdf** utility to combine these PDF files into one PDF file. For example, if you produce three PDF files that represent usage patterns that occur 53%, 32%, and 15% of the time respectively, you can use this command:

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

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

Notes:

- Avoid mixing the PDF files created by different versions or PTF levels of the XL C/C++ compiler.
 - You cannot edit PDF files that are generated by the resulting application. Otherwise, the performance or function of the generated executable application might be affected.
4. Recompile your program using the same compiler options as before, but change **-qpdf1** to **-qpdf2**. In this second compilation, the accumulated profiling information is used to fine-tune the optimizations. The resulting program contains no profiling overhead and runs at full speed.

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

Notes:

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

Examples

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

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

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

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

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

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

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

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

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

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

#Run with one set of input data
./a.out < sample.data

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

Related information in the XL C/C++ Compiler Reference



-qpdf1, -qpdf2



-O, -qoptimize



Runtime environment variables

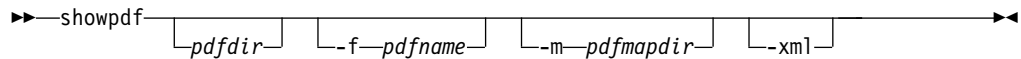
Viewing profiling information with showpdf

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

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

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

Syntax



Parameters

pdfdir

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

pdfname

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

pdfmapdir

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

-xml

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

Usage

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

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

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

Notes:

- PDF and PDF map files must be generated from the same compilation instance. Otherwise, the compiler issues an error.

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

The following example shows how to use the **showpdf** utility to view the profiling information for a Hello World application:

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

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

1. Compile the source file.
`xlc -qpdf1 -O hello.c`
2. Run the resulting executable program **a.out** using a typical data set or several typical data sets.
3. If you want to view the profiling information for the executable file in text format, run the **showpdf** utility without any parameters.

`showpdf`

The result is as follows:

HelloWorld(67): 1 (hello.c)

Call Counters:
4 | 1 printf(69)

Call coverage = 100% (1/1)

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

Block coverage = 100% (2/2)

main(68): 1 (hello.c)

Call Counters:
8 | 1 HelloWorld(67)

Call coverage = 100% (1/1)

Block Counters:
6-9 | 1
10 |

Block coverage = 100% (1/1)

Total Call coverage = 100% (2/2)

Total Block coverage = 100% (3/3)

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

`showpdf -xml`

The result is as follows:

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

Related information in the *XL C/C++ Compiler Reference*



`-qpdf1, -qpdf2`



`-qshowpdf`

Object level profile-directed feedback

About this task

In addition to optimizing entire executables, profile-directed feedback (PDF) can also be applied to specific object files. This can be an advantage in applications where patches or updates are distributed as object files or libraries rather than as executables. Also, specific areas of functionality in your application can be

optimized without the process of relinking the entire application. In large applications, you can save the time and trouble that otherwise need to be spent relinking the application.

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

The steps below outline this process:

1. Compile your program using the **-qpdf1** option. For example:

```
xlc -c -O3 -qpdf1 file1.c file2.c file3.c
```

In this example, we are using the optimization level **-O3** to indicate that we want a moderate level of optimization.

2. Link the object files to get an instrumented executable:

```
xlc -O3 -qpdf1 file1.o file2.o file3.o
```

3. Run the instrumented executable with sample data that is representative of the data you want to optimize for.

```
a.out < sample_data
```

4. Compile the program again using the **-qpdf2** option. Specify the **-qnoipa** option so that the linking step is skipped and PDF optimization is applied to the object files rather than to the entire executable.

```
xlc -c -O3 -qpdf2 -qnoipa file1.c file2.c file3.c
```

The resulting output of this step are object files optimized for the sample data processed by the original instrumented executable. In this example, the optimized object files would be file1.o, file2.o, and file3.o. These can be linked by using the system loader **ld** or by omitting the **-c** option in the **-qpdf2** step.

Notes:

- You must use the same optimization level in all the steps. In this example, the optimization level is **-O3**.
- If you want to specify a file name for the profile that is created, use the **pdfname** suboption in both the **-qpdf1** and **-qpdf2** steps. For example:

```
xlc -O3 -qpdf1=pdfname=myprofile file1.c file2.c file3.c
```

Without the **pdfname** suboption, by default the file name is **._pdf**; the location of the file is the current working directory or whatever directory you have set using the **PDFDIR** environment variable. If the **PDFDIR** environment variable is set but the specified directory does not exist, the compiler issues a warning message.

- Because the **-qnoipa** option needs to be specified in the **-qpdf2** step so that linking of your object files is skipped, you cannot use interprocedural analysis (IPA) optimizations and object level PDF at the same time.

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

Marking variables as local or imported

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

-qdatalocal

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

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

-qdataimported

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

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

Related information in the XL C/C++ Compiler Reference



`-qdataimported`, `-qdatalocal`, `-qtocdata`

Getting the most out of `-qdatalocal`

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

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

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

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

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

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

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

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

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

Notes:

- On 64-bit Linux, TOC entries are pointer size. When you specify `-qdatalocal` without arguments, the option is ignored for variables that are larger than the pointer size. Conversely, data smaller than pointer size is word-aligned. See the following example of an `objdump` excerpt that shows when a char (r3) is marked local. The offset between the byte and the next data (r4) is still 4 bytes. The data is accessed by a load byte instruction instead of a regular load.

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

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

Related information in the XL C/C++ Compiler Reference



`-qdataimported`, `-qdatalocal`, `-qtocdata`

Using compiler reports to diagnose optimization opportunities

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

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

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

Inline reports

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

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

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

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

Loop transformations

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

- Reasons why a loop cannot be automatically parallelized
- Reasons why a loop cannot be unrolled

- Reasons why SIMD vectorization failed

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

Data reorganizations

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

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

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

Profile-directed feedback reports

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

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

Related information

 `-qlistfmt`

Parsing compiler reports with development tools

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

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

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

Other optimization options

Options are available to control particular aspects of optimization. They are often enabled as a group or given default values when you enable a more general optimization option or level.

For more information about these options, see the heading for each option in the *XL C/C++ Compiler Reference*.

Table 18. Selected compiler options for optimizing performance

Option	Description
-qignerrno	Allows the compiler to assume that <code>errno</code> is not modified by library function calls, so that such calls can be optimized. Also allows optimization of square root operations, by generating inline code rather than calling a library function.
-qsmallstack	Instructs the compiler to compact stack storage. Doing so might increase heap usage, which might increase execution time. However, it might be necessary for the program to run or to be optimally multithreaded.
-finline-functions	Controls inlining.
-funroll-loops, -funroll-all-loops	Independently controls loop unrolling. -funroll-all-loops is implicitly activated under -O3 .
 -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 nonvolatile register saves and restores. In C++, the -qnounwind option implies the -qnoeh option. It should not be used if the program uses <code>setjmp/longjmp</code> or any other form of exception handling.
-qstrict	Disables all transformations that change program semantics. In general, compiling a correct program with -qstrict and any levels of optimization produces the same results as without optimization.
-qnostrict	Allows the compiler to reorder floating-point calculations and potentially excepting instructions. A potentially excepting instruction is one that might raise an interrupt due to erroneous execution (for example, floating-point overflow, a memory access violation). -qnostrict is used by default for the -O3 and higher optimization levels.
-qprefetch	Inserts prefetch instructions in compiled code to improve code performance. In situations where you are working with applications that generate a high cache-miss rate, you can use its suboption assistthread to generate prefetching assist threads (for example, <code>-qprefetch=assistthread</code>). -qnoprefetch is the default option.

Related information in the *XL C/C++ Compiler Reference*

-  **-qignerrno**
-  **-qsmallstack**
-  **-finline-functions**
-  **-funroll-loops, -funroll-all-loops**
-  **-qinlglue**
-  **-qeh (C++ only)**

 -qunwind

 -qstrict

 -qprefetch

Chapter 9. Debugging optimized code

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

For example:

Data location issues

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

Instruction scheduling issues

With an optimized program, the compiler might reorder instructions. That is, instructions might not be executed in the order you would expect based on the sequence of lines in the original source code. Also, the sequence of instructions for a statement might not be contiguous. As you step through the program with a debugger, the program might appear as if it is returning to a previously executed line in the code (interleaving of instructions).

Consolidating variable values

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

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

Debug non-optimized code first

Debug a non-optimized version of your program first, and then recompile it with your desired optimization options. See “Debugging in the presence of optimization” on page 60 for some compiler options that are useful in this approach.

Use -g level

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

Understanding different results in optimized programs

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

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

Use with caution debugging techniques that rely on examining values in storage, unless the **-g8** or **-g9** option is in effect and the optimization level is **-O2**. The compiler might have deleted or moved a common expression evaluation. It might have assigned some variables to registers so that they do not appear in storage at all.

Debugging in the presence of optimization

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

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

- qlist** Instructs the compiler to emit an object listing. The object listing includes hex and pseudo-assembly representations of the generated instructions, traceback tables, and text constants.
- qreport** Instructs the compiler to produce a report of the loop transformations it performed, what inlining was done, and some other transformations. To make **-qreport** generate a listing file, you must also specify one of the **-qhot**, **-qsmp**, **-finline-functions**, **-qsimd**, or other optimization options.
- qipa=list** Instructs the compiler to emit an object listing that provides information for IPA optimization.
- qcheck** Generates code that performs certain types of runtime checking.
- qsmp=noopt** If you are debugging SMP code, **-qsmp=noopt** ensures that the compiler

performs only the minimum transformations necessary to parallelize your code and preserves maximum debug capability.

-qkeepparam

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

-qinitauto

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

-g

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

Chapter 10. Coding your application to improve performance

Chapter 8, “Optimizing your applications,” on page 31 discusses the various compiler options that the XL C/C++ compiler provides for optimizing your code with minimal coding effort. If you want to take your application a step further to complement and take the most advantage of compiler optimizations, the following sections discuss C and C++ programming techniques that can improve performance of your code:

- “Finding faster input/output techniques”
- “Reducing function-call overhead”
- “Using template explicit instantiation declarations (C++11)” on page 65
- “Managing memory efficiently (C++ only)” on page 65
- “Optimizing variables” on page 66
- “Manipulating strings efficiently” on page 66
- “Optimizing expressions and program logic” on page 67
- “Optimizing operations in 64-bit mode” on page 68
- “Using rvalue references (C++11)” on page 68
- “Using visibility attributes (IBM extension)” on page 70

Finding faster input/output techniques

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

- If your file I/O accesses do not exhibit locality (that is truly random access such as in a database), implement your own buffering or caching mechanism on the low-level I/O functions.
- If you do your own I/O buffering, make the buffer a multiple of 4KB, which is the minimum size of a page.
- Use buffered I/O to handle text files.
- If you have to process an entire file, determine the size of the data to be read in, allocate a single buffer to read it to, read the whole file into that buffer at once using read, and then process the data in the buffer. This reduces disk I/O, provided the file is not so big that excessive swapping will occur. Consider using the mmap function to access the file.

Reducing function-call overhead

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

- Call a function directly, rather than using function pointers.
- Use const arguments in inlined functions whenever possible. Functions with constant arguments provide more opportunities for optimization.
- Use the restrict keyword for pointers that can never point to the same memory.
- Use **#pragma disjoint** within functions for pointers or reference parameters that can never point to the same memory.
- Declare a nonmember function as static whenever possible. This can speed up calls to the function and increase the likelihood that the function will be inlined.

- **C++** Usually, you should not declare all your 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 they have few parameters and the most frequently used parameters are in the leftmost positions in the function prototype.
- Avoid passing by value large structures or unions as function parameters or returning a large structure or a union. Passing such aggregates requires the compiler to copy and store many values. 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. Homogeneous structs, unions and arrays that meet one of the following conditions can be efficiently passed as value parameters or returned as function results:
 - Contain only up to eight floating-point values of the same type where complex is counted as two.
 - Contain up to eight vector values or values processed in vector registers.
- Pass non-aggregate types such as `int` and `short` or small aggregates by value rather than passing by reference, whenever possible.
- If your function exits by returning the value of another function with the same parameters that were passed to your function, put the parameters in the same order in the function prototypes. The compiler can then branch directly to the other function.
- Use the built-in functions, which include string manipulation, floating-point, and trigonometric functions, instead of coding your own. Intrinsic functions require less overhead and are faster than a function call, and they often allow the compiler to perform better optimization.
- **C++** Many functions from the C++ standard libraries are mapped to optimized built-in functions by the compiler.
- **C** Many functions from `string.h` and `math.h` are mapped to optimized built-in functions by the compiler.
- Selectively mark your functions for inlining using the `inline` keyword. An inlined function requires less overhead and is generally faster than a function call. The best candidates for inlining are small functions that are called frequently from a few places, or functions called with one or more compile-time constant parameters, especially those that affect `if`, `switch`, or `for` statements. You might also want to put these functions into header files, which allows automatic inlining across file boundaries even at low optimization levels. Be sure to inline all functions that only load or store a value, or use simple operators such as comparison or arithmetic operators. Large functions and functions that are called rarely are generally not good candidates for inlining. Neither are medium size functions that are called from many places.
- Avoid breaking your program into too many small functions. If you must use small functions, seriously consider using the `-qipa` compiler option, which can automatically inline such functions, and using other techniques for optimizing calls between functions.

-  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 in the XL C/C++ Compiler Reference



-qisolated_call



#pragma disjoint



-qipa

Using template explicit instantiation declarations (C++11)

Note: IBM supports the majority of C++11 features, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard.

Use the explicit instantiation declarations feature to suppress the implicit instantiation of a template specialization or its members. This helps reduce the collective size of the object files and shorten compilation time.

Managing memory efficiently (C++ only)

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 aligned members first. Members of similar alignment should be grouped together where possible.
- In a structure, place variables near each other if they are frequently used together.
- Ensure that objects that are no longer needed are freed or otherwise made available for reuse. One way to do this is to use an *object manager*. Each time you create an instance of an object, pass the pointer to that object to the object manager. The object manager maintains a list of these pointers. To access an object, you can call an object manager member function to return the information to you. The object manager can then manage memory usage and object reuse.
- Storage pools are a good way of keeping track of used memory (and reclaiming it) without having to resort to an object manager or reference counting. Do not use storage pools for objects with non-trivial destructors, because in most implementations the destructors cannot be run when the storage pool is cleared.
- Avoid copying large and complicated objects.
- Avoid performing a *deep copy* if you only need a *shallow copy*. 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 that are contained within that object, and so on. A deep copy must be performed in multithreaded environments, because it reduces sharing and synchronization.
- Use virtual methods only when absolutely necessary.
- Use the "Resource Acquisition is Initialization" (RAII) pattern.
- Use `shared_ptr` and `weak_ptr`.

Optimizing variables

Consider the following guidelines:

- Use local variables, preferably automatic variables, as much as possible. The compiler must make several worst-case assumptions about global variables. For example, if a function uses external variables and also calls external functions, the compiler assumes that every call to an external function could use and change the value of every external variable. If you know that a global variable is not read or affected by any function call and this variable is read several times with function calls interspersed, copy the global variable to a local variable and then use this local variable.
- If you must use global variables, use static variables with file scope rather than external variables whenever possible. In a file with several related functions and static variables, the optimizer can gather and use more information about how the variables are affected.
- If you must use external variables, group external data into structures or arrays whenever it makes sense to do so. All elements of an external structure use the same base address. Do not group variables whose addresses are taken with variables whose addresses are not taken.
- Avoid taking the address of a variable. If you use a local variable as a temporary variable and must take its address, avoid reusing the temporary variable for a different purpose. Taking the address of a local variable can inhibit optimizations that would otherwise be done on calculations involving that variable.
- Use constants instead of variables where possible. The optimizer is able to do a better job reducing runtime calculations by doing them at compile time instead. For instance, if a loop body has a constant number of iterations, use constants in the loop condition to improve optimization (for (i=0; i<4; i++) can be better optimized than for (i=0; i<x; i++)). An enumeration declaration can be used to declare a named constant for maintainability.
- Use register-sized integers (long data type) for scalars to avoid sign extension instructions after each change. For large arrays of integers, consider using one-byte or two-byte integers or bit fields.
- Use the smallest floating-point precision appropriate to your computation.
- An extern variables that must be shared within a shared library but need not be accessed from outside the library must be declared with a visibility attribute or an option that limits its visibility to the library. This allows it to be accessed directly instead of via the TOC.

Related information in the XL C/C++ Compiler Reference



-qisolated_call

Manipulating strings efficiently

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

- When you store strings into allocated storage, align the start of the string on an 8-byte or 16-byte boundary.
- Keep track of the length of your strings. If you know the length of a string, you can use mem functions instead of str functions. For example, memcpy is faster than strcpy because it does not have to search for the end of the string.
- If you are certain that the source and target do not overlap, use memcpy instead of memmove. This is because memcpy copies directly from the source to the

destination, while `memmove` might copy the source to a temporary location in memory before copying to the destination, or it might copy in reverse order depending on the length of the string.

- When manipulating strings using `mem` functions, faster code can be generated if the `count` parameter is a constant rather than a variable. This is especially true for small count values.
- Make string literals read-only, whenever possible. When the same string is used multiple times, making it read-only improves certain optimization techniques, reduces memory usage, and shortens compilation time. You can explicitly set strings to read-only by using `-qro` (this is enabled by default  except when compiling with `cc` ) to avoid changing your source files.

Related information in the XL C/C++ Compiler Reference



Optimizing expressions and program logic

Consider the following guidelines:

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

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

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

- Do not use global variables as loop indices or bounds.
- Avoid `goto` statements that jump into the middle of loops. Such statements inhibit certain optimizations.
- Improve the predictability of your code by making the fall-through path more probable. Code such as:

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

should be written as:

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

- If one or two cases of a `switch` statement are typically executed much more frequently than other cases, break out those cases by handling them separately before the `switch` statement. If possible, replace the `switch` statement by checking whether the value is in range to be obtained from an array.
-  Use `try` blocks for exception handling only when necessary because they can inhibit optimization.
- Keep array index expressions as simple as possible.

Optimizing operations in 64-bit mode

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

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

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

- Avoid performing mixed 32-bit and 64-bit operations. For example, adding a 32-bit data type to a 64-bit data type requires that the 32-bit be sign-extended to clear or set the upper 32-bit of the register. This slows the computation.
- Use long types instead of signed, unsigned, and plain int types for variables that will be frequently accessed, such as loop counters and array indexes. Doing so frees the compiler from having to truncate or sign-extend array references, parameters during function calls, and function results during returns.

Using rvalue references (C++11)

Note: IBM supports the majority of C++11 features, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard.

In C++11, you can overload functions based on the value categories of arguments and similarly have lvalueness detected by template argument deduction. You can also have an rvalue bound to an rvalue reference and modify the rvalue through the reference. This enables a programming technique with which you can reuse the resources of expiring objects and therefore improve the performance of your libraries, especially if you use generic code with class types, for example, template data structures. Additionally, the value category can be considered when writing a forwarding function.

Move semantics

When you want to optimize the use of temporary values, you can use a move operation in what is known as destructive copying. Consider the following string concatenation and assignment:

```
std::string a, b, c;  
c = a + b;
```

In this program, the compiler first stores the result of `a + b` in an internal temporary variable, that is, an rvalue.

The signature of a normal copy assignment operator is as follows:

```
string& operator = (const string&)
```

With this copy assignment operator, the assignment consists of the following steps:

1. Copy the temporary variable into `c` using a deep-copy operation.
2. Discard the temporary variable.

Deep copying the temporary variable into `c` is not efficient because the temporary variable is discarded at the next step.

To avoid the needless duplication of the temporary variable, you can implement an assignment operator that moves the variable instead of copying the variable. That is, the argument of the operator is modified by the operation. A move operation is faster because it is done through pointer manipulation, but it requires a reference through which the source variable can be manipulated. However, `a + b` is a temporary value, which is not easily differentiated from a `const`-qualified value in C++ before C++11 for the purposes of overload resolution.

With rvalue references, you can create a move assignment operator as follows:

```
string& operator= (string&&)
```

With this move assignment operator, the memory allocated for the underlying C-style string in the result of `a + b` is assigned to `c`. Therefore, it is not necessary to allocate new memory to hold the underlying string in `c` and to copy the contents to the new memory.

The following code can be an implementation of the `string` move assignment operator:

```
string& string::operator=(string&& str)
{
    // The named rvalue reference str acts like an lvalue
    std::swap(_capacity, str._capacity);
    std::swap(_length, str._length);

    // char* _str points to a character array and is a
    // member variable of the string class
    std::swap(_str, str._str);
    return *this;
}
```

However, in this implementation, the memory originally held by the `string` being assigned to is not freed until `str` is destroyed. The following implementation that uses a local variable is more memory efficient:

```
string& string::operator=(string&& parm_str)
{
    // The named rvalue reference parm_str acts like an lvalue
    string sink_str;
    std::swap(sink_str, parm_str);
    std::swap(*this, sink_str);
    return *this;
}
```

In a similar manner, the following program is a possible implementation of a `string` concatenation operator:

```
string operator+(string&& a, const string& b)
{
    return std::move(a+=b);
}
```

Note: The `std::move` function only casts the result of `a+=b` to an rvalue reference, without moving anything. The return value is constructed using a move constructor because the expression `std::move(a+=b)` is an rvalue. The relationship

between a move constructor and a copy constructor is analogous to the relationship between a move assignment operator and a copy assignment operator.

Perfect forwarding

The `std::forward` function is a helper template, much like `std::move`. It returns a reference to its function argument, with the resulting value category determined by the template type argument. In an instantiation of a forwarding function template, the value category of an argument is encoded as part of the deduced type for the related template type parameter. The deduced type is passed to the `std::forward` function.

The wrapper function in the following example is a forwarding function template that forwards to the `do_work` function. Use `std::forward` in forwarding functions on the calls to the target functions. The following example also uses the `decltype` and trailing return type features to produce a forwarding function that forwards to one of the `do_work` functions. Calling the wrapper function with any argument results in a call to a `do_work` function if a suitable overload function exists. Extra temporaries are not created and overload resolution on the forwarding call resolves to the same overload as it would if the `do_work` function were called directly.

```
struct s1 *do_work(const int&);           // #1
struct s2 *do_work(const double&);       // #2
struct s3 *do_work(int&&);               // #3
struct s4 *do_work(double&&);            // #4
template <typename T> auto wrapper(T && a)->
    decltype(do_work(std::forward<T>(*static_cast<typename std::remove_reference<T>
    ::type*>(0))))
{
    return do_work(std::forward<T>(a));
}
template <typename T> void tPtr(T *t);
int main()
{
    int x;
    double y;
    tPtr<s1>(wrapper(x));    // calls #1
    tPtr<s2>(wrapper(y));    // calls #2
    tPtr<s3>(wrapper(0));    // calls #3
    tPtr<s4>(wrapper(1.0));  // calls #4
}
```

Related information in the XL C/C++ Compiler Reference

 `-qlanglvl`

Using visibility attributes (IBM extension)

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

- Decrease the size of shared libraries.
- Reduce the possibility of symbol collision.
- Allow more optimization for the compile and link phases.
- Improve the efficiency of dynamic linking.
- Within a shared library, allow direct access instead of via a TOC pointer.

Supported types of entities

C++

The compiler supports visibility attributes for the following entities:

- Function
- Variable
- Structure/union/class
- Enumeration
- Template
- Namespace

C++

C

The compiler supports visibility attributes for the following entities:

- Function
- Variable

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

C

Related information in the *XL C/C++ Compiler Reference*



-fvisibility



-shared (-qmckshrobj)



#pragma GCC visibility push, #pragma GCC visibility pop

Related information in the *XL C/C++ Language Reference*



The visibility variable attribute (IBM extension)



The visibility function attribute (IBM extension)



The visibility type attribute (C++ only) (IBM extension)



The visibility namespace attribute (C++ only) (IBM extension)

Types of visibility attributes

The following table describes different visibility attributes.

Table 19. *Visibility attributes*

Attribute	Description
default	Indicates that external linkage entities have the default attribute in object files. These entities are exported in shared libraries, and can be preempted.
protected	Indicates that external linkage entities have the protected attribute in object files. These entities are exported in shared libraries, but cannot be preempted.

Table 19. Visibility attributes (continued)

Attribute	Description
hidden	Indicates that external linkage entities have the hidden attribute in object files. These entities are not exported in shared libraries, but their addresses can be referenced indirectly through pointers.
internal	Indicates that external linkage entities have the internal attribute in object files. These entities are not exported in shared libraries, and their addresses are not available to other modules in shared libraries.
Notes: In this release, the hidden and internal visibility attributes are the same. The addresses of the entities that are specified with either of these visibility attributes can be referenced indirectly through pointers.	

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

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

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

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

//b.h
extern void temp();

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

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

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

```
xlc -c -fPIC a.c b.c
xlc -shared -o libtest.so a.o b.o
```

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

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

The output of the example is as follows:

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

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

Rules of visibility attributes

Priority of visibility attributes

The visibility attributes have a priority sequence, which is default < protected < hidden < internal. You can see Example 3 and Example 9 for reference.

Rules of determining the visibility attributes

C

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

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

C

C++

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

1. If the entity has an explicitly specified visibility attribute, the specified visibility attribute takes effect.
2. Otherwise, if the entity is a template instantiation or specialization, and the template has a visibility attribute, the visibility attribute of the entity is propagated from that of the template. See Example 1.
3. Otherwise, if the entity has any of the following enclosing contexts, the visibility attribute of this entity is propagated from that of the nearest context. See Example 2. For the details of propagation rules, see “Propagation rules (C++ only)” on page 78.
 - Structure/class

- Enumeration
- Namespace
- Pragma directives

Restriction: Pragma directives do not affect the visibility attributes of class members and template specializations.

- Otherwise, the visibility attribute of the entity is determined by the following visibility attribute settings. The visibility attribute that has the highest priority is the actual visibility attribute of the entity. See Example 3. For the priority of the visibility attributes, see Priority of visibility attributes.
 - The setting of the **-fvisibility** option.
 - The visibility attribute of the type of the entity, if the entity is a variable and its type has a visibility attribute.
 - The visibility attribute of the return type of the entity, if the entity is a function and its return type has a visibility attribute.
 - The visibility attributes of the parameter types of the entity, if the entity is a function and its parameter types have visibility attributes.
 - The visibility attributes of template arguments or template parameters of the entity, if the entity is a template and its arguments or parameters have visibility attributes.

Example 1

In the following example, template `template<typename T, typename U> B{}` has the protected visibility attribute. The visibility attribute is propagated to those of template specialization `template<> class B<char, char>{}`, partial specialization `template<typename T> class B<T, float>{}`, and all the types of template instantiations.

```
class __attribute__((visibility("internal"))) A{ vis_v_a;    //internal

//protected
template<typename T, typename U>
class __attribute__((visibility("protected"))) B{
    public:
        void func(){}
};

//protected
template<>
class B<char, char>{
    public:
        void func(){}
};

//protected
template<typename T>
class B<T, float>{
    public:
        void func(){}
};

B<int, int> a;    //protected
B<A, int> b;      //protected
B<char, char> c;  //protected
B<int, float> d;  //protected
B<A, float> e;    //protected

int main(){
    a.func();
}
```

```

        b.func();
        c.func();
        d.func();
        e.func();
    }

```

Example 2

In the following example, the nearest enclosing context of function `func()` is class `B`, so the visibility attribute of `func()` is propagated from that of class `B`, which is hidden. The nearest enclosing context of class `A` is the pragma directives whose setting is protected, so the visibility of class `A` is protected.

```

namespace __attribute__((visibility("internal"))) ns{
#pragma GCC visibility push(protected)
    class A{
        class __attribute__((visibility("hidden"))) B{
            int func(){};
        };
    };
#pragma GCC visibility pop
};

```

Example 3

In the following example, the visibility attribute specified by the **-fvisibility** option is protected. The type of variable `vis_v_d` is class `CD`, whose visibility attribute is default. The visibility attribute that has a higher priority of these two attributes is protected, so the actual visibility attribute of variable `vis_v_d` is protected. The same rule applies to the determination of the visibility attributes of variables `vis_v_p`, `vis_v_h`, and `vis_v_i`. For functions `vis_f_fun1`, `vis_f_fun2`, and `vis_f_fun3`, their visibility attributes are determined by those of their parameter types, return types, and the setting of the **-fvisibility** option. For template functions `vis_f_template1` and `vis_f_template2`, their visibility attributes are determined by those of their template arguments, template parameters, function parameter types, return types, and the setting of the **-fvisibility** option. The visibility attribute that has the highest priority takes effect.

```

//The -fvisibility=protected option is specified
class __attribute__((visibility("default"))) CD {} vis_v_d; //protected
class __attribute__((visibility("protected"))) CP {} vis_v_p; //protected
class __attribute__((visibility("hidden"))) CH {} vis_v_h; //hidden
class __attribute__((visibility("internal"))) CI {} vis_v_i; //internal

void vis_f_fun1(CH a, CP b, CD c, CI d) {} //internal
void vis_f_fun2(CD a) {} //protected
CH vis_f_fun3(CI a, CP b) {} //internal

template<class T, class U> T vis_f_template1(T t, U u){}
template<class T, int N> void vis_f_template2(T t, int i){}

int main(){
    vis_f_template1<CD, CH>(vis_v_d, vis_v_p); //hidden
    vis_f_template2<CD, 10>(vis_v_p, 10); // protected
}

```

C++

Rules and restrictions of using the visibility attributes

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

- You can specify visibility attributes only for entities that have external linkage. The compiler issues a warning message when you set the visibility attribute for entities with other linkages, and the specified visibility attribute is ignored. See Example 4.
- You cannot specify different visibility attributes in the same declaration or definition of an entity; otherwise, the compiler issues an error message. See Example 5.
- If an entity has more than one declaration that is specified with different visibility attributes, the visibility attribute of the entity is the first visibility attribute that the compiler processes. See Example 6.
- You cannot specify visibility attributes in the typedef statements. See Example 7.
-  If type T has a visibility attribute, types T*, T&, and T&& have the same visibility attribute with that of type T. See Example 8.
-  If a class and its enclosing classes do not have explicitly specified visibilities and the visibility attribute of the class has a lower priority than those of its nonstatic member types and its bases classes, the compiler issues a warning message. See Example 9. For the priority of the visibility attributes, see Priority of visibility attributes. 
-  The visibility attribute of a namespace does not apply for the namespace with the same name. See Example 10. 
-  If you specify a visibility attribute for a global new or delete operator, the compiler issues a warning message to ignore the visibility attribute unless the visibility attribute is default. See Example 11. 

Example 4

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

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

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

Example 5

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

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

Example 6

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

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

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

Example 7

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

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

➤ C++

Example 8

In this example, the visibility attribute of class `CP` is protected, so the visibility attribute of `CP*` and `CP&` is also protected.

```
class __attribute__((visibility("protected"))) CP {} vis_v_p;
class CP* vis_v_p_p = &vis_v_p; //protected
class CP& vis_v_lr_p = vis_v_p; //protected
```

Example 9

In this example, the compiler accepts the default visibility attribute of class `Derived1` because the visibility attribute is explicitly specified for class `Derived1`. The compiler also accepts the protected visibility attribute of class `Derived2` because the visibility attribute is propagated from that of the enclosing class `A`. Class `Derived3` does not have an explicitly specified visibility attribute or an enclosing class, and its visibility attribute is default. The compiler issues a warning message because the visibility attribute of class `Derived3` has a lower priority than those of its parent class `Base` and the nonstatic member function `fun()`.

```
//The -fvisibility=default option is specified.
//base class
struct __attribute__((visibility("hidden"))) Base{
    int vis_f_fun(){
        return 0;
    }
};

//Ok
struct __attribute__((visibility("default"))) Derived1: public Base{
    int vis_f_fun(){
        return Base::vis_f_fun();
    }
};
vis_v_d;

//Ok
struct __attribute__((visibility("protected"))) A{
    struct Derived2: public Base{
        int vis_f_fun(){
            __attribute__((visibility("protected")))
        };
    }
};

//Warning
```

```
struct Derived3: public Base{
    //Warning
    int fun() __attribute__((visibility("protected"))){};
};
```

Example 10

In this example, the visibility attribute of the definition of namespace X does not apply to the extension of namespace X.

```
//The -fvisibility=default option is specified.
//namespace definition
namespace X __attribute__((visibility("protected"))){
    int a; //protected
    int b; //protected
}
//namespace extension
namespace X {
    int c; //default
    int d; //default
}
//equivalent to namespace X
namespace Y {
    int __attribute__((visibility("protected"))) a; //protected
    int __attribute__((visibility("protected"))) b; //protected
    int c; //default
    int d; //default
}
```

Example 11

In this example, the new and delete operators defined outside of class A are global functions, so the explicitly specified hidden visibility attribute does not take effect. The new and delete operations defined within class A are local ones, so you can specify visibility attributes for them.

```
#include <stddef.h>
//default
void* operator new(size_t) throw (std::bad_alloc) __attribute__((visibility("hidden")))
{
    return 0;
};
void operator delete(void*) throw () __attribute__((visibility("hidden"))){}

class A{
public:
    //hidden
    void* operator new(size_t) throw (std::bad_alloc) __attribute__((visibility("hidden")))
    {
        return 0;
    };
    void operator delete(void*) throw () __attribute__((visibility("hidden"))){}
};
```

C++

Propagation rules (C++ only)

Visibility attributes can be propagated from one entity to other entities. The following table lists all the cases for visibility propagation.

Table 20. Propagation of visibility attributes

Original entity	Destination entities	Example
Namespace	Named namespaces that are defined in the original namespace	<pre>namespace A __attribute__((visibility("hidden"))){ // Namespace B has the hidden visibility attribute, // which is propagated from namespace A. namespace B{} // The unnamed namespace does not have a visibility // attribute. namespace{} }</pre>
Namespace	Classes that are defined in the original namespace	<pre>namespace A __attribute__((visibility("hidden"))){ // Class B has the hidden visibility attribute, // which is propagated from namespace A. class B; // Object x has the hidden visibility attribute, // which is propagated from namespace A. class{} x; }</pre>
Namespace	Functions that are defined in the original namespace	<pre>namespace A __attribute__((visibility("hidden"))){ // Function fun() has the hidden visibility // attribute, which is propagated from namespace A. void fun(){}; }</pre>
Namespace	Objects that are defined in the original namespace	<pre>namespace A __attribute__((visibility("hidden"))){ // Variable m has the hidden visibility attribute, // which is propagated from namespace A. int m; }</pre>
Class	Member classes	<pre>class __attribute__((visibility("hidden"))) A{ // Class B has the hidden visibility attribute, // which is propagated from class A. class B{}; }</pre>
Class	Member functions or static member variables	<pre>class __attribute__((visibility("hidden"))) A{ // Function fun() has the hidden visibility // attribute, which is propagated from class A. void fun(){}; // Static variable m has the hidden visibility // attribute, which is propagated from class A. static int m; }</pre>

Table 20. Propagation of visibility attributes (continued)

Original entity	Destination entities	Example
Template	Template instantiations/ template specifications/ template partial specializations	<pre> template<typename T, typename U> class __attribute__((visibility("hidden"))) A{ public: void fun(){}; }; // Template instantiation class A<int, char> has the // hidden visibility attribute, which is propagated // from template class A(T,U). class A<int, char>{ public: void fun(){}; }; // Template specification // template<> class A<double, double> has the hidden // visibility attribute, which is propagated // from template class A(T,U). template<> class A<double, double>{ public: void fun(){}; }; // Template partial specification // template<typename T> class A<T, char> has the // hidden visibility attribute, which is propagated // from template class A(T,U). template<typename T> class A<T, char>{ public: void fun(){}; }; </pre>
Template argument/ parameter	Template instantiations/ template specifications/ template partial specializations	<pre> template<typename T> void fun1(){} template<typename T> void fun2(T){} class M __attribute__((visibility("hidden"))){} m; // Template instantiation fun1<M>() has the hidden // visibility attribute, which is propagated from // template argument M. fun1<M>(); // Template instantiation fun2<M>(M) has the hidden // visibility attribute, which is propagated from // template parameter m. fun2(m); // Template specification fun1<M>() has the hidden // visibility attribute, which is propagated from // template argument M. template<> void fun1<M>(); </pre>
Inline function	Static local variables	<pre> inline void __attribute__((visibility("hidden"))) fun(){ // Variable m has the hidden visibility attribute, // which is propagated from inline function fun(). static int m = 4; } </pre>

Table 20. Propagation of visibility attributes (continued)

Original entity	Destination entities	Example
Type	Entities of the original type	<pre>class __attribute__((visibility("hidden"))) A {};</pre> // Object x has the hidden visibility attribute, // which is propagated from class A. <pre>class A x;</pre>
Function return type	Function	<pre>class __attribute__((visibility("hidden"))) A{};</pre> // Function fun() has the hidden visibility attribute, // which is propagated from function return type A. <pre>A fun();</pre>
Function parameter type	Function	<pre>class __attribute__((visibility("hidden"))) A{};</pre> // Function fun(class A) has the hidden visibility // attribute, which is propagated from function // parameter type A. <pre>void fun(class A);</pre>

Specifying visibility attributes using the -fvisibility option

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

Specifying visibility attributes using pragma preprocessor directives

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

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

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

➤ **C++** Visibility pragma directives #pragma GCC visibility push and #pragma GCC visibility pop affect only namespace-scope declarations. Class members and template specializations are not affected. See Example 3 and Example 4. **C++** ◀

Examples

Example 1

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

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

```

        int vis_v_j;           //protected
#       pragma GCC visibility push(hidden)
        int vis_v_k;           //hidden
#       pragma GCC visibility pop
#       pragma GCC visibility pop
#       pragma GCC visibility pop
}
#pragma GCC visibility pop

```

Example 2

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

```

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

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

```

➤ C++

Example 3

In this example, the visibility attribute of class members `vis_v_i` and `vis_f_fun()` is hidden. The visibility attribute is propagated from that of the class, but is not affected by the pragma directives.

```

class __attribute__((visibility("hidden"))) A{
#pragma GCC visibility push(protected)
public:
    static int vis_v_i;
    void vis_f_fun(){}
#pragma GCC visibility pop
} vis_v_a;

```

Example 4

In this example, the visibility attribute of function `vis_f_fun()` is hidden. The visibility attribute is propagated from that of the template specialization or partial specialization, but is not affected by the pragma directives.

```

namespace ns{
#pragma GCC visibility push(hidden)
    template <typename T, typename U> class TA{
    public:
        void vis_f_fun(){}
    };
#pragma GCC visibility pop

#pragma GCC visibility push(protected)
    //The visibility attribute of the template specialization is hidden.
    template <> class TA<char, char>{
    public:
        void vis_f_fun(){}
    };
#pragma GCC visibility pop

#pragma GCC visibility push(default)
    //The visibility attribute of the template partial specialization is hidden.
    template <typename T> class TA<T, long>{

```

```
    public:  
    void vis_f_fun(){}  
};  
#pragma GCC visibility pop
```

C++

Chapter 11. Using the high performance libraries

IBM XL C/C++ for Linux, V13.1.2 is shipped with a set of libraries for high-performance mathematical computing:

- The Mathematical Acceleration Subsystem (MASS) is a set of libraries of tuned mathematical intrinsic functions that provide improved performance over the corresponding standard system math library functions. MASS is described in “Using the Mathematical Acceleration Subsystem libraries (MASS).”
- The Basic Linear Algebra Subprograms (BLAS) are a set of routines that provide matrix/vector multiplication functions tuned for PowerPC architectures. The BLAS functions are described in “Using the Basic Linear Algebra Subprograms – BLAS” on page 96.

Using the Mathematical Acceleration Subsystem libraries (MASS)

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

The MASS libraries consist of a library of scalar C/C++ functions described in “Using the scalar library” on page 86, a set of vector libraries tuned for specific architectures described in “Using the vector libraries” on page 88, and a set of SIMD libraries tuned for specific architectures described in “Using the SIMD libraries” on page 92. The functions contained in both scalar and vector libraries are automatically called at certain levels of optimization, but you can also call them explicitly in your programs. Note that accuracy and exception handling might not be identical in MASS functions and system library functions.

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

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

- **-qhot -qignerrno -qnostrict**
- **-qhot -qignerrno -qstrict=nolib**
- **-qhot -O3**
- **-O4**
- **-O5**

the compiler automatically attempts to vectorize calls to system math functions by calling the equivalent MASS vector functions (with the exceptions of functions `vdnint`, `vdint`, `vcosin`, `vscosin`, `vqdr`, `vsqdr`, `vrqdr`, `vsrqdr`, `vpopcnt4`, `vpopcnt8`, `vexp2`, `vexp2m1`, `vsexp2`, `vsexp2m1`, `vlog2`, `vlog2lp`, `vslog2`, and `vslog2lp`). If it cannot vectorize, it automatically tries to call the equivalent MASS scalar functions. For automatic vectorization or scalarization, the compiler uses versions of the MASS functions contained in the XLOPT library `libxlopt.a`.

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

“Compiling and linking a program with MASS” on page 95 describes how to compile and link a program that uses the MASS libraries, and how to selectively use the MASS scalar library functions in conjunction with the regular system libraries.

Related external information

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

Using the scalar library

The MASS scalar library `libmass.a` contains an accelerated set of frequently used math intrinsic functions that provide improved performance over the corresponding standard system library functions. The MASS scalar functions are used when explicitly linking `libmass.a`.

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

1. Provide the prototypes for the functions by including `math.h` and `mass.h` in your source files.
2. Link the MASS scalar library with your application. For instructions, see “Compiling and linking a program with MASS” on page 95.

The MASS scalar functions accept double-precision parameters and return a double-precision result, or accept single-precision parameters and return a single-precision result, except `sincos` which gives 2 double-precision results. They are summarized in Table 21.

Table 21. MASS scalar functions

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
<code>acos</code>	<code>acosf</code>	Returns the arccosine of x	<code>double acos (double x);</code>	<code>float acosf (float x);</code>
<code>acosh</code>	<code>acoshf</code>	Returns the hyperbolic arccosine of x	<code>double acosh (double x);</code>	<code>float acoshf (float x);</code>
	<code>anint</code>	Returns the rounded integer value of x		<code>float anint (float x);</code>
<code>asin</code>	<code>asinf</code>	Returns the arcsine of x	<code>double asin (double x);</code>	<code>float asinf (float x);</code>
<code>asinh</code>	<code>asinhf</code>	Returns the hyperbolic arcsine of x	<code>double asinh (double x);</code>	<code>float asinhf (float x);</code>
<code>atan2</code>	<code>atan2f</code>	Returns the arctangent of x/y	<code>double atan2 (double x, double y);</code>	<code>float atan2f (float x, float y);</code>
<code>atan</code>	<code>atanf</code>	Returns the arctangent of x	<code>double atan (double x);</code>	<code>float atanf (float x);</code>
<code>atanh</code>	<code>atanhf</code>	Returns the hyperbolic arctangent of x	<code>double atanh (double x);</code>	<code>float atanhf (float x);</code>
<code>cbrt</code>	<code>cbrtf</code>	Returns the cube root of x	<code>double cbrt (double x);</code>	<code>float cbrtf (float x);</code>
<code>copysign</code>	<code>copysignf</code>	Returns x with the sign of y	<code>double copysign (double x, double y);</code>	<code>float copysignf (float x);</code>
<code>cos</code>	<code>cosf</code>	Returns the cosine of x	<code>double cos (double x);</code>	<code>float cosf (float x);</code>

Table 21. MASS scalar functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
cosh	coshf	Returns the hyperbolic cosine of x	double cosh (double x);	float coshf (float x);
cosisin		Returns a complex number with the real part the cosine of x and the imaginary part the sine of x.	double_Complex cosisin (double);	
dnint		Returns the nearest integer to x (as a double)	double dnint (double x);	
erf	erff	Returns the error function of x	double erf (double x);	float erff (float x);
erfc	erfcf	Returns the complementary error function of x	double erfc (double x);	float erfcf (float x);
exp	expf	Returns the exponential function of x	double exp (double x);	float expf (float x);
expm1	expm1f	Returns (the exponential function of x) - 1	double expm1 (double x);	float expm1f (float x);
hypot	hypotf	Returns the square root of $x^2 + y^2$	double hypot (double x, double y);	float hypotf (float x, float y);
lgamma	lgammaf	Returns the natural logarithm of the absolute value of the Gamma function of x	double lgamma (double x);	float lgammaf (float x);
log	logf	Returns the natural logarithm of x	double log (double x);	float logf (float x);
log10	log10f	Returns the base 10 logarithm of x	double log10 (double x);	float log10f (float x);
log1p	log1pf	Returns the natural logarithm of (x + 1)	double log1p (double x);	float log1pf (float x);
pow	powf	Returns x raised to the power y	double pow (double x, double y);	float powf (float x, float y);
rsqrt		Returns the reciprocal of the square root of x	double rsqrt (double x);	
sin	sinf	Returns the sine of x	double sin (double x);	float sinf (float x);
sincos		Sets *s to the sine of x and *c to the cosine of x	void sincos (double x, double* s, double* c);	
sinh	sinhf	Returns the hyperbolic sine of x	double sinh (double x);	float sinhf (float x);
sqrt		Returns the square root of x	double sqrt (double x);	
tan	tanf	Returns the tangent of x	double tan (double x);	float tanf (float x);
tanh	tanhf	Returns the hyperbolic tangent of x	double tanh (double x);	float tanhf (float x);

Notes:

- The trigonometric functions (sin, cos, tan) return NaN (Not-a-Number) for large arguments (where the absolute value is greater than $2^{50}\pi$).
- In some cases, the MASS functions are not as accurate as the ones in the `libm.a` library, and they might handle edge cases differently (`sqrt(Inf)`, for example).
- For accuracy comparisons with `libm.a`, see Product documentation (manuals) in the Product support content section of the Mathematical Acceleration Subsystem website.
-

Related external information

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

Using the vector libraries

If you want to explicitly call any of the MASS vector functions, you can do so by including `massv.h` in your source files and linking your application with the appropriate vector library. Information about linking is provided in “Compiling and linking a program with MASS” on page 95.

libmassv.a

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

libmassvp8.a

Contains functions that have been tuned for the POWER8 architecture.

The single-precision and double-precision floating-point functions contained in the vector libraries are summarized in Table 22 on page 89. The integer functions contained in the vector libraries are summarized in Table 23 on page 91. 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 in the following paragraph), all of the floating-point functions in the vector libraries accept three parameters:

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

The functions are of the form

function_name (*y,x,n*)

where *y* is the target vector, *x* is the source vector, and *n* is the vector length. The parameters *y* and *x* are assumed to be double-precision for functions with the prefix `v`, and single-precision for functions with the prefix `vs`. As an example, the following code:

```
#include <massv.h>

double x[500], y[500];
```

```

int n;
n = 500;
...
vexp (y, x, &n);

```

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

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

In `vcosisin(y, x, n)` and `vscosisin(y, x, n)`, x is a vector of n elements and the function outputs a vector y of n `__Complex` elements of the form $(\cos(x[i]), \sin(x[i]))$.

Table 22. MASS floating-point vector functions

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
<code>vacos</code>	<code>vsacos</code>	Sets $y[i]$ to the arc cosine of $x[i]$, for $i=0,\dots,n-1$	<code>void vacos (double y[], double x[], int *n);</code>	<code>void vsacos (float y[], float x[], int *n);</code>
<code>vacosh</code>	<code>vsacosh</code>	Sets $y[i]$ to the hyperbolic arc cosine of $x[i]$, for $i=0,\dots,n-1$	<code>void vacosh (double y[], double x[], int *n);</code>	<code>void vsacosh (float y[], float x[], int *n);</code>
<code>vasin</code>	<code>vsasin</code>	Sets $y[i]$ to the arc sine of $x[i]$, for $i=0,\dots,n-1$	<code>void vasin (double y[], double x[], int *n);</code>	<code>void vsasin (float y[], float x[], int *n);</code>
<code>vasinh</code>	<code>vsasinh</code>	Sets $y[i]$ to the hyperbolic arc sine of $x[i]$, for $i=0,\dots,n-1$	<code>void vasinh (double y[], double x[], int *n);</code>	<code>void vsasinh (float y[], float x[], int *n);</code>
<code>vatan2</code>	<code>vsatan2</code>	Sets $z[i]$ to the arc tangent of $x[i]/y[i]$, for $i=0,\dots,n-1$	<code>void vatan2 (double z[], double x[], double y[], int *n);</code>	<code>void vsatan2 (float z[], float x[], float y[], int *n);</code>
<code>vatanh</code>	<code>vsatanh</code>	Sets $y[i]$ to the hyperbolic arc tangent of $x[i]$, for $i=0,\dots,n-1$	<code>void vatanh (double y[], double x[], int *n);</code>	<code>void vsatanh (float y[], float x[], int *n);</code>
<code>vcbrt</code>	<code>vscbrt</code>	Sets $y[i]$ to the cube root of $x[i]$, for $i=0,\dots,n-1$	<code>void vcbrt (double y[], double x[], int *n);</code>	<code>void vscbrt (float y[], float x[], int *n);</code>
<code>vcos</code>	<code>vscos</code>	Sets $y[i]$ to the cosine of $x[i]$, for $i=0,\dots,n-1$	<code>void vcos (double y[], double x[], int *n);</code>	<code>void vscos (float y[], float x[], int *n);</code>
<code>vcosh</code>	<code>vscosh</code>	Sets $y[i]$ to the hyperbolic cosine of $x[i]$, for $i=0,\dots,n-1$	<code>void vcosh (double y[], double x[], int *n);</code>	<code>void vscosh (float y[], float x[], int *n);</code>
<code>vcosisin</code>	<code>vscosisin</code>	Sets the real part of $y[i]$ to the cosine of $x[i]$ and the imaginary part of $y[i]$ to the sine of $x[i]$, for $i=0,\dots,n-1$	<code>void vcosisin (double _Complex y[], double x[], int *n);</code>	<code>void vscosisin (float _Complex y[], float x[], int *n);</code>
<code>vdint</code>		Sets $y[i]$ to the integer truncation of $x[i]$, for $i=0,\dots,n-1$	<code>void vdint (double y[], double x[], int *n);</code>	

Table 22. MASS floating-point vector functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
vdiv	vsdiv	Sets $z[i]$ to $x[i]/y[i]$, for $i=0,...,*n-1$	void vdiv (double z[], double x[], double y[], int *n);	void vsdiv (float z[], float x[], float y[], int *n);
vdnint		Sets $y[i]$ to the nearest integer to $x[i]$, for $i=0,...,*n-1$	void vdnint (double y[], double x[], int *n);	
verf	vserf	Sets $y[i]$ to the error function of $x[i]$, for $i=0,...,*n-1$	void verf (double y[], double x[], int *n)	void vs erf (float y[], float x[], int *n)
verfc	vserfc	Sets $y[i]$ to the complimentary error function of $x[i]$, for $i=0,...,*n-1$	void verfc (double y[], double x[], int *n)	void vs erf c (float y[], float x[], int *n)
vexp	vsexp	Sets $y[i]$ to the exponential function of $x[i]$, for $i=0,...,*n-1$	void vexp (double y[], double x[], int *n);	void vs exp (float y[], float x[], int *n);
vexp2	vsexp2	Sets $y[i]$ to 2 raised to the power of $x[i]$, for $i=1,...,*n-1$	void vexp2 (double y[], double x[], int *n);	void vs exp2 (float y[], float x[], int *n);
vexpm1	vsexpm1	Sets $y[i]$ to (the exponential function of $x[i]$)-1, for $i=0,...,*n-1$	void vexpm1 (double y[], double x[], int *n);	void vs expm1 (float y[], float x[], int *n);
vexp2m1	vsexp2m1	Sets $y[i]$ to (2 raised to the power of $x[i]$) - 1, for $i=1,...,*n-1$	void vexp2m1 (double y[], double x[], int *n);	void vs exp2m1 (float y[], float x[], int *n);
vhypot	vshypot	Sets $z[i]$ to the square root of the sum of the squares of $x[i]$ and $y[i]$, for $i=0,...,*n-1$	void vhypot (double z[], double x[], double y[], int *n)	void vshypot (float z[], float x[], float y[], int *n)
vlog	vslog	Sets $y[i]$ to the natural logarithm of $x[i]$, for $i=0,...,*n-1$	void vlog (double y[], double x[], int *n);	void vs log (float y[], float x[], int *n);
vlog2	vslog2	Sets $y[i]$ to the base-2 logarithm of $x[i]$, for $i=1,...,*n-1$	void vlog2 (double y[], double x[], int *n);	void vs log2 (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 vs log10 (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 vs log1p (float y[], float x[], int *n);
vlog21p	vslog21p	Sets $y[i]$ to the base-2 logarithm of $(x[i]+1)$, for $i=1,...,*n-1$	void vlog21p (double y[], double x[], int *n);	void vs log21p (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 vs pow (float z[], float x[], float y[], int *n);
vqdrft	vsqdrft	Sets $y[i]$ to the fourth root of $x[i]$, for $i=0,...,*n-1$	void vqdrft (double y[], double x[], int *n);	void vs qdrft (float y[], float x[], int *n);

Table 22. MASS floating-point vector functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
vrcbrt	vsrbrt	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 vsrbrt (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);
vrqdrft	vsrqdrft	Sets $y[i]$ to the reciprocal of the fourth root of $x[i]$, for $i=0,...,n-1$	void vrqdrft (double y[], double x[], int *n);	void vsrqdrft (float y[], float x[], int *n);
vsqrt	vsrsqrt	Sets $y[i]$ to the reciprocal of the square root of $x[i]$, for $i=0,...,n-1$	void vsqrt (double y[], double x[], int *n);	void vsrsqrt (float y[], float x[], int *n);
vsin	vssin	Sets $y[i]$ to the sine of $x[i]$, for $i=0,...,n-1$	void vsin (double y[], double x[], int *n);	void vssin (float y[], float x[], int *n);
vsincos	vssincos	Sets $y[i]$ to the sine of $x[i]$ and $z[i]$ to the cosine of $x[i]$, for $i=0,...,n-1$	void vsincos (double y[], double z[], double x[], int *n);	void vssincos (float y[], float z[], float x[], int *n);
vsinh	vssinh	Sets $y[i]$ to the hyperbolic sine of $x[i]$, for $i=0,...,n-1$	void vsinh (double y[], double x[], int *n);	void vssinh (float y[], float x[], int *n);
vsqrt	vssqrt	Sets $y[i]$ to the square root of $x[i]$, for $i=0,...,n-1$	void vsqrt (double y[], double x[], int *n);	void vssqrt (float y[], float x[], int *n);
vtan	vstan	Sets $y[i]$ to the tangent of $x[i]$, for $i=0,...,n-1$	void vtan (double y[], double x[], int *n);	void vstan (float y[], float x[], int *n);
vtanh	vstanh	Sets $y[i]$ to the hyperbolic tangent of $x[i]$, for $i=0,...,n-1$	void vtanh (double y[], double x[], int *n);	void vstanh (float y[], float x[], int *n);

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 23. MASS integer vector library functions

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

Overlap of input and output vectors

In most applications, the MASS vector functions are called with disjoint input and output vectors; that is, the two vectors do not overlap in memory. Another common usage scenario is to call them with the same vector for both input and output parameters (for example, `vsin (y, y, &n)`). Other kinds of overlap (where

input and output vectors are neither disjoint nor identical) should be avoided, since they might 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 might be obtained.
- For calls to vector functions that take two input vectors (for example, `vatan2 (y, x1, x2, &n)`):
The previous restriction applies to both pairs of vectors `y,x1` and `y,x2`. That is, `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.

Alignment of input and output vectors

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

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.

Related information in the XL C/C++ Compiler Reference



Related external information

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

Using the SIMD libraries

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

1. Provide the prototypes for the functions by including `mass_simd.h` in your source files.
2. Link the MASS SIMD library `libmass_simdp8.a` with your application. For instructions, see “Compiling and linking a program with MASS” on page 95.

The single-precision MASS SIMD functions accept single-precision arguments and return single-precision results. Likewise, the double-precision MASS SIMD functions accept double-precision arguments and return double-precision results. They are summarized in Table 24 on page 93.

Table 24. MASS SIMD functions

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
acosd2	acosf4	Computes the arc cosine of each element of vx.	vector double acosd2 (vector double vx);	vector float acosf4 (vector float vx);
acoshd2	acoshf4	Computes the arc hyperbolic cosine of each element of vx.	vector double acoshd2 (vector double vx);	vector float acoshf4 (vector float vx);
asind2	asinf4	Computes the arc sine of each element of vx.	vector double asind2 (vector double vx);	vector float asinf4 (vector float vx);
asinhd2	asinhf4	Computes the arc hyperbolic sine of each element of vx.	vector double asinhd2 (vector double vx);	vector float asinhf4 (vector float vx);
atand2	atanf4	Computes the arc tangent of each element of vx.	vector double atand2 (vector double vx);	vector float atanf4 (vector float vx);
atan2d2	atan2f4	Computes the arc tangent of each element of vx/vy.	vector double atan2d2 (vector double vx, vector double vy);	vector float atan2f4 (vector float vx, vector float vy);
atanhd2	atanhf4	Computes the arc hyperbolic tangent of each element of vx.	vector double atanhd2 (vector double vx);	vector float atanhf4 (vector float vx);
cbrtd2	cbrtf4	Computes the cube root of each element of vx.	vector double cbrtd2 (vector double vx);	vector float cbrtf4 (vector float vx);
cosd2	cosf4	Computes the cosine of each element of vx.	vector double cosd2 (vector double vx);	vector float cosf4 (vector float vx);
coshd2	coshf4	Computes the hyperbolic cosine of each element of vx.	vector double coshd2 (vector double vx);	vector float coshf4 (vector float vx);
cosisind2	cosisinf4	Computes the cosine and sine of each element of x, and stores the results in y and z as follows: cosisind2 (x,y,z) sets y and z to {cos(x1), sin(x1)} and {cos(x2), sin(x2)} where x={x1,x2}. cosisinf4 (x,y,z) sets y and z to {cos(x1), sin(x1), cos(x2), sin(x2)} and {cos(x3), sin(x3), cos(x4), sin(x4)} where x={x1,x2,x3,x4}.	void cosisind2 (vector double x, vector double *y, vector double *z)	void cosisinf4 (vector float x, vector float *y, vector float *z)
divd2	divf4	Computes the quotient vx/vy.	vector double divd2 (vector double vx, vector double vy);	vector float divf4 (vector float vx, vector float vy);

Table 24. MASS SIMD functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
erfcd2	erfcf4	Computes the complementary error function of each element of vx.	vector double erfcd2 (vector double vx);	vector float erfcf4 (vector float vx);
erfd2	erff4	Computes the error function of each element of vx.	vector double erfd2 (vector double vx);	vector float erff4 (vector float vx);
expd2	expf4	Computes the exponential function of each element of vx.	vector double expd2 (vector double vx);	vector float expf4 (vector float vx);
exp2d2	exp2f4	Computes 2 raised to the power of each element of vx.	vector double exp2d2 (vector double vx);	vector float exp2f4 (vector float vx);
expm1d2	expm1f4	Computes (the exponential function of each element of vx) - 1.	vector double expm1d2 (vector double vx);	vector float expm1f4 (vector float vx);
exp2m1d2	exp2m1f4	Computes (2 raised to the power of each element of vx) -1.	vector double exp2m1d2 (vector double vx);	vector float exp2m1f4 (vector float vx);
hypotd2	hypotf4	For each element of vx and the corresponding element of vy, computes $\sqrt{x^2+y^2}$.	vector double hypotd2 (vector double vx, vector double vy);	vector float hypotf4 (vector float vx, vector float vy);
lgammad2	lgammaf4	Computes the natural logarithm of the absolute value of the Gamma function of each element of vx .	vector double lgammad2 (vector double vx);	vector float lgammaf4 (vector float vx);
logd2	logf4	Computes the natural logarithm of each element of vx.	vector double logd2 (vector double vx);	vector float logf4 (vector float vx);
log2d2	log2f4	Computes the base-2 logarithm of each element of vx.	vector double log2d2 (vector double vx);	vector float log2f4 (vector float vx);
log10d2	log10f4	Computes the base-10 logarithm of each element of vx.	vector double log10d2 (vector double vx);	vector float log10f4 (vector float vx);
log1pd2	log1pf4	Computes the natural logarithm of each element of (vx +1).	vector double log1pd2 (vector double vx);	vector float log1pf4 (vector float vx);
log21pd2	log21pf4	Computes the base-2 logarithm of each element of (vx +1).	vector double log21pd2 (vector double vx);	vector float log21pf4 (vector float vx);
powd2	powf4	Computes each element of vx raised to the power of the corresponding element of vy.	vector double powd2 (vector double vx, vector double vy);	vector float powf4 (vector float vx, vector float vy);

Table 24. MASS SIMD functions (continued)

Double-precision function	Single-precision function	Description	Double-precision function prototype	Single-precision function prototype
qdrtd2	qdrtf4	Computes the quad root of each element of vx.	vector double qdrtd2 (vector double vx);	vector float qdrtf4 (vector float vx);
rcbrtd2	rcbrtf4	Computes the reciprocal of the cube root of each element of vx.	vector double rcbrtd2 (vector double vx);	vector float rcbrtf4 (vector float vx);
recipd2	recipf4	Computes the reciprocal of each element of vx.	vector double recipd2 (vector double vx);	vector float recipf4 (vector float vx);
rqdrtd2	rqdrtf4	Computes the reciprocal of the quad root of each element of vx.	vector double rqdrtd2 (vector double vx);	vector float rqdrtf4 (vector float vx);
rsqrtd2	rsqrtf4	Computes the reciprocal of the square root of each element of vx.	vector double rsqrtd2 (vector double vx);	vector float rsqrtf4 (vector float vx);
sincosd2	sincosf4	Computes the sine and cosine of each element of vx.	void sincosd2 (vector double vx, vector double *vs, vector double *vc);	void sincosf4 (vector float vx, vector float *vs, vector float *vc);
sind2	sinf4	Computes the sine of each element of vx.	vector double sind2 (vector double vx);	vector float sinf4 (vector float vx);
sinhd2	sinhf4	Computes the hyperbolic sine of each element of vx.	vector double sinhd2 (vector double vx);	vector float sinhf4 (vector float vx);
sqrtd2	sqrtf4	Computes the square root of each element of vx.	vector double sqrtd2 (vector double vx);	vector float sqrtf4 (vector float vx);
tand2	tanf4	Computes the tangent of each element of vx.	vector double tand2 (vector double vx);	vector float tanf4 (vector float vx);
tanhd2	tanhf4	Computes the hyperbolic tangent of each element of vx.	vector double tanhd2 (vector double vx);	vector float tanhf4 (vector float vx);

Compiling and linking a program with MASS

To compile an application that calls the functions in the following MASS libraries, specify the corresponding library names on the **-l** link option.

Table 25. The scalar, vector, and SIMD MASS library

MASS library	Library name
Scalar library	mass
Vector library	massv or massvp8
SIMD library	mass_simdp8

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

Link object file `prog.c` with scalar library `libmass.a` and vector library `libmassv.a`

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

Link object file `prog.c` with SIMD library `libmass_simdp8.a`

```
xlc prog.c -o prog -lmass_simdp8
```

Using `libmass.a` with the math system library

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

1. Use the `ar` command to extract the object files of the wanted functions from `libmass.a`. For most functions, the object file name is the function name followed by `.s64.o`.¹ For example, to extract the object file for the `tan` function, the command would be:

```
ar -x tan.s64.o libmass.a
```

2. Archive the extracted object files into another library:

```
ar -qv libfasttan.a tan.s64.o
ranlib libfasttan.a
```

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

```
xlc sample.c -o sample -Ldir_containing_libfasttan -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.

Exceptions:

1. The `sin` and `cos` functions are both contained in the object file `sincos.s64.o`. The `cosisin` and `sincos` functions are both contained in the object file `cosisin.s64.o`.
2. The XL C/C++ `pow` function is contained in the object file `dx.s64.o`.

Note: The `cos` and `sin` functions will both be exported if either one is exported. `cosisin` and `sincos` will both be exported if either one is exported.

Using the Basic Linear Algebra Subprograms – BLAS

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

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

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

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

“BLAS function syntax” on page 97 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 information and examples of usage of these functions, see *Engineering and Scientific Subroutine Library Guide and Reference*, available at the Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL web page.

“Linking the libxlopt library” on page 99 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 input matrix *a*, where:

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

m represents:

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

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

n represents:

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

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

alpha

is the scaling constant for matrix *a*

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

lda

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

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

incx

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

beta

is the scaling constant for vector *y*

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

incy

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

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

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

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

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

The parameters are as follows:

transa

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

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

transb

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

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

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

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

m represents:

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

and:

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

m must be greater than or equal to zero.

alpha

is the scaling constant for matrix *a*

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

lda

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

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

ldb

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

beta

is the scaling constant for matrix *c*

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

ldc

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

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

Linking the libxlopt library

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

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

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

Chapter 12. Parallelizing your programs

The compiler offers you the following methods of implementing shared memory program parallelization:

- Automatic parallelization of countable program loops, which are defined in “Countable loops.” An overview of the compiler's automatic parallelization capabilities is provided in “Enabling automatic parallelization” on page 103.
- 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 105.

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.

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

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

Related information:

“Using shared-memory parallelism (SMP)” on page 41

Related information in the *XL C/C++ Compiler Reference*



XLSMPOPTS



OpenMP environment variables for parallel processing

Related external information



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

Countable loops

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

Countable for loop syntax with single statement

```

▶▶for( [iteration_variable] ; —exit_condition— ; —increment_expression— ) —————▶
▶statement—————▶▶

```

Countable for loop syntax with statement block

```

▶▶for( [iteration_variable] ; [expression] ) —————▶
▶{ [declaration_list] [statement_list] —increment_expression— [statement_list] } —————▶▶

```

Countable while loop syntax

```

▶▶while( —exit_condition— ) —————▶
▶{ [declaration_list] [statement_list] —increment_expression— } —————▶▶

```

Countable do while loop syntax

```

▶▶do { [declaration_list] [statement_list] —increment_expression— } while ( —exit_condition— ) —————▶▶

```

The following definitions apply to these syntax diagrams:

iteration_variable

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

exit_condition

takes the following form:

```

| —increment_variable— | <= | expression |
|                       | <  |
|                       | >= |
|                       | >  |

```

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

increment_expression

takes any of the following forms:

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

- $iteration_variable = iteration_variable - increment$

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

Enabling automatic parallelization

The compiler can automatically locate and parallelize all countable loops where possible in your program code. A loop is considered to be countable if it has any of the forms shown in “Countable loops” on page 101, 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.

Data sharing attribute rules

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

Data sharing attribute rules for variables referenced in a construct

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

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

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

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

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

- For variables with static storage duration, if they are declared in a scope inside the construct, they are shared.

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

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

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

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

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

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

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

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

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

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

- If variables with static storage duration are declared in called routines in the region, the variables are shared.
- Variables with `const`-qualified types are shared if they have no mutable member and are declared in called routines.
- File-scope or namespace-scope variables referenced in called routines in the region are shared unless they are specified in a `threadprivate` directive.
- Objects with dynamic storage duration are shared.
- Static data members are shared unless they are specified in a `threadprivate` directive.

- The formal arguments of called routines in the region that are passed by reference inherit the data sharing attributes of the associated actual arguments.
- Other variables declared in called routines in the region are private.

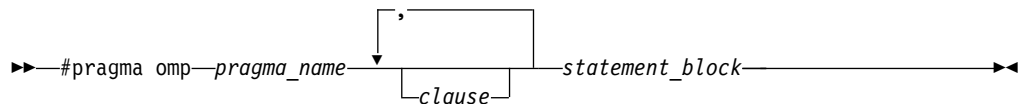
Using OpenMP directives

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

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

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

OpenMP directive syntax



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

The **#pragma omp** pragmas generally appear immediately before the section of code to which they apply. The following example defines a parallel region in which iterations of a for loop can run in parallel:

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

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

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

```

#pragma omp section
    structured_block_2
    ...
    ....
}
}

```

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

Related information in the *XL C/C++ Compiler Reference*



Pragma directives for parallel processing



OpenMP built-in functions



OpenMP environment variables for parallel processing

Shared and private variables in a parallel environment

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

Data scope attribute clause	Description
private	The private clause declares the variables in the list to be private to each thread in a team.
firstprivate	The firstprivate clause provides a superset of the functionality provided by the private clause. The private variable is initialized by the original value of the variable when the parallel construct is encountered.
lastprivate	The lastprivate clause provides a superset of the functionality provided by the private clause. The private variable is updated after the end of the parallel construct.
shared	The shared clause declares the variables in the list to be shared among all the threads in a team. All threads within a team access the same storage area for shared variables.
reduction	The reduction clause performs a reduction on the scalar variables that appear in the list, with a specified operator.
default	The default clause allows the user to affect the data-sharing attribute of the variables appeared in the parallel construct.

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

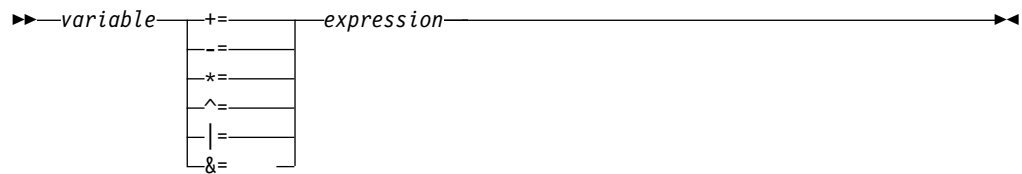
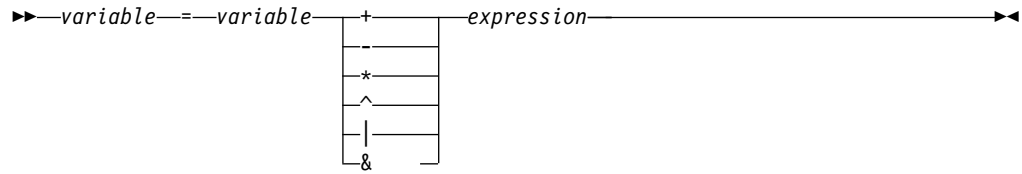
Related information in the *XL C/C++ Compiler Reference*



Pragma directives for parallel processing

Reduction operations in parallelized loops

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



where:

variable

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

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

expression

is any valid expression.

OpenMP directives provide you with mechanisms to specify reduction variables explicitly.

Chapter 13. Vector element order toggling

To consistently use the instructions generated by vector built-in functions, users need to make all existing Vector Multimedia Extension (VMX) and Vector Scalar Extension (VSX) load and store built-in functions operate on the vectors in registers in the same vector element order, either little endian or big endian element order.

Vector element order

The `-maltivec` or `-qaltivec` option affects the vector element order only in registers when the vectors are operated by a specific set of functions. In registers, the vector layout differs when the computer loads the vector in either big endian element order or little endian element order.

Big endian element order

Vectors are laid out in vector registers from left to right, so that element 0 is the leftmost element in the register.

Little endian element order

Vectors are laid out in vector registers from right to left, so that element 0 is the rightmost element in the register.

For more information, see “Example: Vector layout in the memory and register” on page 110.

Rules for vector element order toggling

The vector element order is toggled in registers by following these rules:

- The `-maltivec` or `-qaltivec` option does not affect the vector element order in memory, where the vector elements are always stored in big endian element order.

For example, in memory, the vector initialization is not affected by the `-maltivec` or `-qaltivec` option. The vectors initialized by the union with the non-vectors (such as arrays) are always in big endian element order in memory. When the initialized vector is loaded to registers, the vector element order is always reversed to little endian element order in registers even when `-qaltivec=be`. However, if the vector loading is realized by using the vector built-in function, the vector element order is arranged with respect to the `-maltivec` or `-qaltivec` option.

The previous rules apply to the vector literals as well. The vector literals are stored always in big endian element order in the memory. When the vector literals are loaded to registers, the vector element order is always reversed to little endian element order in registers.

- When `-qaltivec=le` is in effect, the behaviours of functions are as follows:
 - The VMX and VSX load built-in functions load vectors to registers in little endian element order.
 - The VMX and VSX store built-in functions assume that the vectors to be stored are in little endian element order in registers.
 - The nonload and nonstore built-in functions assume that vectors are loaded in registers in little endian element order.

- When `-qaltivec=be` is in effect, these functions operate on the vectors in an opposite way of `-qaltivec=le`. The vectors in registers are in big endian element order.
- Regardless of the `-maltivec` or `-qaltivec` option, the `vec_xl_be` function loads vectors to registers always in big endian element order and the `vec_xst_be` function assumes that vectors to be stored are always in big endian element order in registers.

For more information, see “Example: The vector built-in functions affected by the `-maltivec` or `-qaltivec` option” on page 111 and “Example: The vector initialization by using the union with arrays” on page 111.

Example: Vector layout in the memory and register

The following example gets the first element of vector `va` by calling the `vec_extract` function. The function returning value is different based on the `-maltivec` or `-qaltivec` option that determines whether `vec_extract` arranges the vector elements in big endian or little endian element order.

```
int get_first_element(va)
{
    vector signed int va;
    printf("%i\n", vec_extract(va, 0));
    //vec_extract is affected by the -maltivec or -qaltivec option
}
```

The following tables show the vector layout in the memory and the register.

Table 26. Vector layout in the memory

Vector element value	E0	E1	E2	E3
----------------------	----	----	----	----

- When `-qaltivec=be`, the vector elements are loaded to registers in big endian element order and vector layout looks as follows.

Table 27. Vector layout in big endian element order

Vector element number	0	1	2	3
Vector element value	E0	E1	E2	E3

The elements of vector `va` are ordered from the first to last, and stored from the left of registers. The `get_first_element` function gets the first element E0 from the left of registers.

- When `-qaltivec=le`, the vector elements are loaded to registers in little endian element order and vector layout looks as follows.

Table 28. Vector layout in little endian element order

Vector element number	3	2	1	0
Vector element value	E3	E2	E1	E0

The elements of vector `va` are ordered from the last to first, and also stored from the left of registers. The `get_first_element` function gets the first element E0 from the right of registers.

Example: The vector built-in functions affected by the `-maltivec` or `-qaltivec` option

The following program `vec_xlw4.c` shows that the `vec_xlw4` function loads the vector elements in registers in the order specified by the `-maltivec` or `-qaltivec` option.

```
int main()
{
    vector signed int a4;
    int c4[4] = {0,1,2,3};
    a4 = vec_xlw4(0, c4);
    //vec_xlw4 is affected by the -maltivec or -qaltivec option
    printf("%i %i %i %i\n", a4[0], a4[1], a4[2], a4[3]);
}
```

- Compile the program with `-qaltivec=be` by running the following command:

```
xlc vec_xlw4.c -qaltivec=be
```

Output:

```
0 1 2 3
```

- Compile the program with `-qaltivec=le` by running the following command:

```
xlc vec_xlw4.c -qaltivec=le
```

Output:

```
0 1 2 3
```

Example: The vector initialization by using the union with arrays

The following program example `vec_equiv.c` contains the vectors initialization by using the union with arrays. The vector loading is not affected by the `-maltivec` or `-qaltivec` option and is loaded to registers always in little endian element order. Therefore, the vector elements extracted by `vec_extract` function are different between `-qaltivec=be` and `-qaltivec=le`.

```
int main()
{
    union {
        vector signed int a4;
        int c4[4];
    };

    //In the memory, the vector initialization (by using the union)
    //is not affected by the -maltivec or -qaltivec option and the vector is stored in
    //big endian element order. Then, the initialized vector is loaded
    //in registers by being reversed to the little endian element order.

    c4[0] = 0; c4[1] = 1; c4[2] = 2; c4[3] = 3;
    for (int i=0; i<4; i++)
        printf("%i ", vec_extract(a4,i));
    //vec_extract is affected by the -maltivec or -qaltivec option

    printf("\n");
}
```

- Compile the codes with `-qaltivec=le` by running the following command:

```
xlc vec_equiv.c -qaltivec=le
```

Output:

```
0 1 2 3
```

- Compile the program with `-qaltivec=be` by running the following command:

```
xlc vec_equiv.c -qaltivec=be
```

Output:

3 2 1 0

The compilation result is different from that of compilation with `-qaltivec=le`.

Related information:



`-maltivec (-qaltivec)`

Program migration from big endian systems

When migrating the programs that contain the Vector Multimedia Extension (VMX) and Vector Scalar Extension (VSX) built-in functions from big endian systems, you can use `-qaltivec=be` to minimize program changes, but you need to pay attention in specific cases.

The following table shows what users need to pay attention when migrating codes from big endian systems by using `-qaltivec=be`.

Table 29. Attention when `-qaltivec=be`

Case	Attention
If the existing program contains only VMX load and store built-in functions	Using <code>-qaltivec=be</code> may affect the program performance; using <code>-qaltivec=le</code> may affect the performance in different ways.
If the existing program contains only VSX load and store built-in functions	In the existing programs, you can use the <code>vec_xl</code> and <code>vec_xst</code> functions to replace the VSX load and store built-in functions to maximally simplify the code changes.
If the existing program contains both VMX and VSX load and store built-in functions	You need to pay attention to the differences of the element order of vectors that are operated by the VMX and VSX built-in functions in little endian systems.
If the existing program contains the vector initialization by using union with arrays	You need to use the <code>vec_ld</code> or <code>vec_xl</code> function to load the vectors explicitly, instead of using the union with arrays, or you can reverse the element order of the array used for vector initialization.
Vector literals	Based on the meaning and usage of vector literals, the user must change the code properly.

Related information:



`-maltivec (-qaltivec)`



Vector built-in functions

Notices

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

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

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

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

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

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

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

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

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

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

websites. The materials at those websites are not part of the materials for this IBM product and use of those websites 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 want 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:

Intellectual Property Dept. for Rational Software
IBM Corporation
5 Technology Park Drive
Westford, MA 01886
U.S.A.

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.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

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

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1998, 2015.

PRIVACY POLICY CONSIDERATIONS:

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, or to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at <http://www.ibm.com/privacy> and IBM's Online Privacy Statement at <http://www.ibm.com/privacy/details> in the section entitled "Cookies, Web Beacons and Other Technologies," and the "IBM Software Products and Software-as-a-Service Privacy Statement" at <http://www.ibm.com/software/info/product-privacy>.

Trademarks

IBM, the IBM logo, and [ibm.com](http://www.ibm.com) are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at <http://www.ibm.com/legal/copytrade.shtml>.

Adobe is a registered trademark of Adobe Systems Incorporated in the United States, other countries, or both.

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

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

Index

Special characters

- fpack-struct (-qalign) 11
- ftrapping-math (-qflttrap) 20
- mcpu 38
- mcpu (-qarch) 38
- mtune 38
- O0 32
- O2 33
- O3 35
 - trade-offs 35
- O4 36
 - trade-offs 37
- O5 37
 - trade-offs 38
- qcache 36, 38
- qfloat 18, 20
 - IEEE conformance 18
 - multiply-add operations 17
- qhot 39
- qipa 36, 38, 43
 - IPA process 37
- qlistfmt compiler option 55
- qlongdouble
 - corresponding Fortran types 6
- qpdf 45
- qpriority 26
- qsmp 41, 101, 103
- qstrict 18, 35
- qwarn64 1
- shared 25
- y 18

Numerics

- 64-bit mode 4
 - alignment 4
 - bit-shifting 3
 - data types 1
 - Fortran 4
 - long constants 2
 - long types 1
 - optimization 68
 - pointers 3

A

- advanced optimization 34
- aggregate
 - alignment 4, 11, 12
 - Fortran 8
- aligned attribute 14
 - alignment 4, 11
 - bit-fields 13
 - modes 11
 - modifiers 14
- architecture
 - optimization 38
- arrays, Fortran 8
- attribute
 - aligned 14

- attribute (*continued*)
 - init_priority 26
 - packed 14

B

- basic example, described ix
- basic optimization 32
- bit-field 13
 - alignment 13
- bit-shifting 3
- BLAS library 96

C

- C++ templates 23
- C++11
 - delegating constructors 21
 - explicit instantiation declarations 65
 - rvalue references 68
 - target constructors 21
- cloning, function 38, 43
- constants
 - folding 18
 - long types 2
 - rounding 18

D

- Data scoping rules 103
- Data sharing attribute rules 103
- data types 1
 - 64-bit mode 1
 - Fortran 4, 6
 - long 1
 - size and alignment 11
- debugging 59
- dynamic library 25

E

- errors, floating-point 20
- exceptions, floating-point 20

F

- floating-point
 - exceptions 20
 - folding 18
 - IEEE conformance 18
 - range and precision 17
 - rounding 18
- folding, floating-point 18
- Fortran
 - 64-bit mode 4
 - aggregates 8
 - arrays 8
 - data types 4, 6
 - function calls 8

- Fortran (*continued*)
 - function pointers 9
 - identifiers 5
- function calls
 - Fortran 8
 - optimizing 63
- function cloning 38, 43
- function pointers, Fortran 9

H

- hardware optimization 38

I

- IEEE conformance 18
- init_priority attribute 26
- initialization order of C++ static objects 26
- input/output
 - optimizing 63
- interlanguage calls 8
- interprocedural analysis (IPA) 43

L

- libmass library 86
- libmassv library 88
- library
 - BLAS 96
 - MASS 85
 - scalar 86
 - shared (dynamic) 25
 - static 25
 - vector 88
- linear algebra functions 96
- long constants, 64-bit mode 2
- long data type, 64-bit mode 1
- loop optimization 39, 101

M

- marking variables as local or imported 53
- MASS libraries 85
 - scalar functions 86
 - vector functions 88
- matrix multiplication functions 96
- memory
 - management 65
- mergepdf 45
- move 68
- multithreading 41, 101

O

- OpenMP 41, 106, 107
- OpenMP directives 105
- optimization 63

- optimization (*continued*)
 - O0 32
 - O2 33
 - O3 35
 - O4 36
 - O5 37
 - 64-bit mode 68
 - across program units 43
 - advanced 34
 - architecture 38, 53
 - basic 32
 - debugging 59
 - hardware 38
 - loop 39
 - loops 101
 - math functions 85
- optimization and tuning
 - optimizing 31
 - tuning 31
- optimization trade-offs
 - O3 35
 - O4 37
 - O5 38
- optimization, diagnostics 55, 56
- optimizing
 - applications 31

P

- packed attribute 14
- parallelization 41, 101
 - automatic 103
 - OpenMP directives 105
- perfect forwarding 68
- performance tuning 63
- pointers
 - 64-bit mode 3
 - Fortran 9
- pragma
 - omp 105
 - pack 14
 - priority 26
- precision, floating-point numbers 17
- priority of static objects 26
- profile-directed feedback (PDF) 45
- profiling 45

R

- range, floating-point numbers 17
- rounding, floating-point 18

S

- scalar MASS library 86
- shared (dynamic) library 25
- shared memory parallelism (SMP) 41, 101, 105, 106, 107
- showpdf 45
- static library 25
- static objects, C++ 26
- strings
 - optimizing 66
- structure alignment 12
 - 64-bit mode 4

T

- template model 23
- Toggling the vector element order
 - Migrating codes from big endian systems 112
- tuning for performance 38

V

- Vector element order 109
- Vector element order toggling 109
- vector MASS library 88
- visibility attributes 70
 - propagation 78

X

- xlopt library 96
- XML report schema 56



Product Number: 5765-J08; 5725-C73

Printed in USA

SC27-6560-01

