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

Getting Started with XL C/C++

SC09-8015-00

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

Getting Started with XL C/C++

SC09-8015-00

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 35.

First Edition (September 2005)

This edition applies to IBM® XL C/C++ Advanced Edition V8.0 for Linux™ (Program 5724-M16) 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.

IBM welcomes your comments. You can send your comments electronically to the network ID listed below. Be sure to include your entire network address if you wish a reply.

- Internet: compinfo@ca.ibm.com

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

Contents

About this document

Getting Started with XL C/C++ provides a general overview of the XL C/C++ compiler, its more significant features, and how those features can help you improve your software development productivity.

For the benefit of current XL C/C++ users upgrading to this release, *Getting Started with XL C/C++* also includes a summary of features that are new or improved for V8.0.

Getting Started with XL C/C++ is intended only to help familiarize you with the compiler. For detailed information on using the XL C/C++ compiler, you will want to refer to other books in the XL C/C++ Advanced Edition V8.0 for Linux library, described in "IBM XL C/C++ publications" on page viii.

Who should read this document

Getting Started with XL C/C++ is intended for anyone who plans to work with IBM® XL C/C++ Advanced Edition V8.0 for Linux, who is familiar with the Linux® operating system, and who has some previous C and C++ programming experience.

How to use this document

If you are new to XL C/C++ , you should view Chapter 1, "Overview of XL C/C++ features," on page 1 to familiarize yourself with the key features of XL C/C++ and how to begin using it to develop your applications.

If you are already an experienced XL C/C++ user and are now upgrading to the latest release of XL C/C++ , you may want to go directly to Chapter 2, "What's new for V8.0," on page 9 to review that latest changes and feature enhancements to the compiler.

The remaining sections of this guide provide a brief overview of basic program development tasks with XL C/C++.

How this document is organized

This guide includes these topics:

- Chapter 1, "Overview of XL C/C++ features," on page 1 outlines the the key features of the XL C/C++ compiler
- Chapter 2, "What's new for V8.0," on page 9 describes new and updated features offered by the latest version of XL C/C++.
- Chapter 3, "Setting up and customizing XL C/C++," on page 17 provides brief overview information on the steps involved in setting up and customizing XL C/C++, together with pointers on where you can find more detailed information.
- Chapter 4, "Editing, compiling, and linking programs with XL C/C++," on page 19 discusses the basic steps involved in creating and compiling your applications with XL C/C++.

- Chapter 5, "Running XL C/C++ programs," on page 27 describes how to run your compiled applications, including setting of run time options.
- Chapter 6, "XL C/C++ compiler diagnostic aids," on page 29 offers guidance on how to use XL C/C++ compiler diagnostic aids to identify and correct compilation problems with your applications.

Conventions and terminology used in this document

Typographical conventions

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

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Commands, executable names, pragma directives, and compiler options.	Use the **-qmkshrobj** compiler option to create a shared object from the generated object files.
italics	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	Make sure that you update the *size* parameter if you return more than the *size* requested.
`monospace`	Programming keywords and library functions, compiler built-in functions, file and directory names, examples of program code, command strings, or user-defined names.	If you call `omp_destroy_lock` with an uninitialized lock variable, the result of the call is undefined.

How to read syntax diagrams

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

You must enter punctuation marks, parentheses, arithmetic operators, and other special characters as part of the syntax.

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

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

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

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

  The ──►◄ symbol indicates the end of a command, directive, or statement.

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

```
►►──keyword──required_item──────────────────────────────────────────────────►◄
```

- Optional items are shown below the main path.

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

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

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

```
►►──keyword──┬─required_choice1─┬──────────────────────────────────►◄
             └─required_choice2─┘
```

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

```
►►──keyword──┬──────────────────┬──────────────────────────────────►◄
             ├─optional_choice1─┤
             └─optional_choice2─┘
```

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

```
             ┌─default_item───┐
►►──keyword──┴─alternate_item─┴────────────────────────────────────►◄
```

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

```
             ┌───────────────────┐
             ▼                   │
►►──keyword────repeatable_item───┴─────────────────────────────────►◄
```

  A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.
- Keywords are shown in nonitalic letters and should be entered exactly as shown (for example, `extern`).

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

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

```
1  2    3       4       5       6                              9 10
►►─#──pragma──comment──(───────compiler─────────────────────────)─►◄
                        |                                 |
                        +───────date──────────────────────+
                        |                                 |
                        +───────timestamp─────────────────+
                        |                                 |
                        +──+──copyright──+──+─────────────+
                           |             |  |             |
                           +──user───────+  +──,─"characters"─+

                                                 7 8
```

1 This is the start of the syntax diagram.

2 The symbol # must appear first.

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

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

5 An opening parenthesis must be present.

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

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

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

9 A closing parenthesis is required.

10 This is the end of the syntax diagram.

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

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

Examples

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

Related information

IBM XL C/C++ publications

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

- Readme files

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

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

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

  publib.boulder.ibm.com/infocenter/lnxpcomp/index.jsp
- PDF documents

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

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

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

Table 2. XL C/C++ PDF files

Document title	PDF file name	Description
IBM XL C/C++ Advanced Edition V8.0 for Linux Installation Guide, GC09-8017-00	install.pdf	Contains information for installing XL C/C++ and configuring your environment for basic compilation and program execution.
IBM XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference, SC09-8013-00	compiler.pdf	Contains information about the various compiler options, pragmas, macros, environment variables, and built-in functions, including those used for parallel processing.
IBM XL C/C++ Advanced Edition V8.0 for Linux Language Reference, SC09-8016-00	language.pdf	Contains information about the C and C++ programming languages, as supported by IBM, including language extensions for portability and conformance to non-proprietary standards.
IBM XL C/C++ Advanced Edition V8.0 for Linux Programming Guide, SC09-8014-00	proguide.pdf	Contains information on advanced programming topics, such as application porting, interlanguage calls with Fortran code, library development, application optimization and parallelization, and the XL C/C++ high-performance libraries.

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

www.adobe.com

Additional documentation

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

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

Technical support

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

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

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

compinfo@ca.ibm.com

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

www.ibm.com/software/awdtools/xlcpp

How to send your comments

Your feedback is important in helping to provide accurate and high-quality information. If you have any comments about this document or any other XL C/C++ documentation, send your comments by e-mail to:

compinfo@ca.ibm.com

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

Chapter 1. Overview of XL C/C++ features

XL C/C++ Advanced Edition V8.0 for Linux can be used for large, complex, computationally intensive programs, including interlanguage calls with Fortran programs. This section discusses the features of the XL C/C++ compiler at a high level. It is intended for people who are evaluating XL C/C++ and for new users who want to find out more about the product.

Commonality with other XL compilers

XL C/C++, together with XL C and XL Fortran, comprise the family of XL compilers.

The XL compilers are part of a larger family of IBM C, C++, and Fortran compilers that are derived from a common code base that shares compiler function and optimization technologies among a variety of platforms and programming languages, such as AIX, Linux distributions, OS/390, OS/400, z/OS, and z/VM operating systems. The common code base, along with compliance to international programming language standards, helps ensure consistent compiler performance and ease of program portability across multiple operating systems and hardware platforms.

The XL compilers are available for use on AIX and selected Linux distributions.

Documentation, online help, and technical support

This guide provides an overview of XL C/C++ and its features. You can also find more extensive product documentation in the following formats:

- Readme files.
- Installable man pages.
- An HTML-based information system.
- Portable Document Format (PDF) documents.
- Online technical support over the Web.

For more information about product documentation and technical support provided with XL C/C++, see:

- "IBM XL C/C++ publications" on page viii
- "Additional documentation" on page ix
- "Technical support" on page ix

Hardware and operating system support

XL C/C++ Advanced Edition V8.0 for Linux supports several Linux distributions. See the README file and System prerequisites in the *XL C/C++ Advanced Edition V8.0 for Linux Installation Guide* for a complete list of supported distributions and requirements.

The compiler, its libraries, and its generated object programs will run on all POWER3™, POWER4™, POWER5™, POWER5+™, PowerPC®, and PowerPC 970 systems with the required software and disk space.

To take maximum advantage of different hardware configurations, the compiler provides a number of options for performance tuning based on the configuration of the machine used for executing an application.

Highly configurable compiler

XL C/C++ offers you a wealth of features to let you tailor the compiler to your own unique compilation requirements.

Compiler invocation commands

XL C/C++ provides several different commands that you can use to invoke the compiler, for example, **xlC**, **xlc++**, and **xlc**. Each invocation command is unique in that it instructs the compiler to tailor compilation output to meet a specific language level specification. Compiler invocation commands are provided to support all standardized C and C++ language levels, and many popular language extensions as well.

The compiler also provides corresponding ″**_r**″ versions of most invocation commands, for example, **xlC_r**. These ″**_r**″ invocations instruct the compiler to link and bind object files to thread-safe components and libraries, and produce threadsafe object code for compiler-created data and procedures.

For more information about XL C/C++ compiler invocation commands, see "Compiling with XL C/C++" on page 20 in this book or Invoking the compiler or a compiler component in the *XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference*.

Compiler options

You can control the actions of the compiler through a large set of provided compiler options. Different categories of options help you to debug your applications, optimize and tune application performance, select language levels and extensions for compatibility with programs from other platforms, and do many other common tasks that would otherwise require changing the source code.

XL C/C++ lets you specify compiler options through a combination of environment variables, compiler configuration files, command line options, and compiler directive statements embedded in your C or C++ program source.

For more information about XL C/C++ compiler options, see Compiler options reference in the *XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference*.

Custom compiler configuration files

The installation process creates a default compiler configuration file at /etc/opt/ibmcmp/vac/8.0/vac.cfg. This configuration file contains several stanzas that define compiler option default settings.

Your compilation needs may frequently call for specifying compiler option settings other than the defaults settings provided by XL C/C++. If so, XL C/C++ provides the **vac_configure** utility that you can use to create additional configuration files. You can then modify these files with any text editor to contain your own frequently-used compiler option settings.

See Customizing the configuration file in the *XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference* for more information on creating and using custom configuration files.

Language standards compliance

The compiler supports the following programming language specifications for C and C++:

- ISO/IEC 9899:1999 (C99)
- ISO/IEC 9899:1990 (referred to as C89)
- ISO/IEC 14882:2003 (referred to as *Standard C++*)
- ISO/IEC 14882:1998, the first official specification of the language (referred to as C++98)

In addition to the standardized language levels, XL C/C++ also supports language extensions, including:

- OpenMP extensions to support parallelized programming.
- Language extensions to support VMX vector programming.
- A subset of GNU C and C++ language extensions.

See Supported language standards in the *XL C/C++ Advanced Edition V8.0 for Linux Language Reference* for more information about C and C++ language specifications and extensions.

Compatibility with GNU

XL C/C++ supports a subset of the GNU compiler command options to facilitate porting applications developed with **gcc** and **g++**.

This support is available when the **gxlc** or **gxlc++** invocation command is used together with select GNU compiler options. The compiler maps these options to their XL C/C++ compiler option counterparts before invoking the compiler.

The **gxlc** and **gxlc++** invocation commands use the /etc/opt/ibmcmp/vac/8.0/gxlc.cfg plain text configuration file to control GNU-to-XL C/C++ option mappings and defaults. You can customize the /etc/opt/ibmcmp/vac/8.0/gxlc.cfg file to better meet the needs of any unique compilation requirements you may have. See Reusing GNU C/C++ compiler options with gxlc and gxlc++ in the *XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference* for more information.

XL C/C++ uses GNU C and GNU C++ header files together with the GNU C and C++ runtime libraries to produce code that is binary-compatible with that produced with the GNU compiler, GCC Version 3.3. Portions of an application can be built with XL C/C++ and combined with portions built with GCC to produce an application that behaves as if it had been built solely with GCC.

Source-code migration and conformance checking

XL C/C++ helps protect your investment in your existing C and C++ source code by providing compiler invocation commands that instruct the compiler to compile your application to a specific language level. You can also use the **-qlanglvl** compiler option to specify a given language level, and the compiler will issue warnings, errors and severe error messages if language or language extension elements in your program source do not conform to that language level.

See **-qlanglvl** in the *XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference* for more information.

Libraries

XL C/C++ Advanced Edition V8.0 for Linux ships with the following libraries:

- SMP Runtime Library supports both explicit and automated parallel processing.
- Mathematics Acceleration Subsystem (MASS) library of tuned mathematical intrinsic functions, for 32-bit and 64-bit modes.
- Basic Linear Algebra Subsystem (BLAS) library of tuned algebraic functions.
- C++ C++ Runtime Library contains support routines needed by the compiler.

Mathematics Acceleration Subsystem libraries

The IBM Mathematics Acceleration Subsystem (MASS) libraries consist of highly tuned scalar and vector mathematical intrinsic functions tuned specifically for optimum performance on PowerPC processor architectures. You can choose MASS libraries to support high-performance computing on a broad range of processors, or you can select libraries tuned to specific processor families.

The MASS libraries support both 32-bit and 64-bit compilation modes, are thread-safe, and offer improved performance over their corresponding `libm` routines. They are called automatically when you request specific levels of optimization for your application. You can also make explicit calls to MASS functions regardless of whether optimization options are in effect or not.

See Using the Mathematical Acceleration Subsystem in the *XL C/C++ Advanced Edition V8.0 for Linux Programming Guide* for more information.

Basic Linear Algebra Subprograms

The BLAS set of high-performance algebraic functions are shipped in the libxlopt library. These functions let you:

- Compute the matrix-vector product for a general matrix or its transpose.
- Perform combined matrix multiplication and addition for general matrices or their transposes.

For more information about using the BLAS functions, see Using the Basic Linear Algebra Subprograms in the *XL C/C++ Advanced Edition V8.0 for Linux Programming Guide*.

Tools and utilities

xlc_install
This interactive utility helps you install XL C/C++ on your system.

new_install
After you install XL C/C++, running this utility will configure the compiler for use on your system.

vac_configure
Use this utility to create additional compiler configuration files that you can then modify to contain your own custom sets of compiler option default settings.

cleanpdf Command
A command related to profile-directed feedback, used for managing the PDFDIR directory. Removes all profiling information from the specified directory, the PDFDIR directory, or the current directory.

mergepdf Command
A command related to profile-directed feedback (PDF) that provides the ability to weight the importance of two or more PDF records when combining them into a single record. The PDF records must be derived from the same executable.

resetpdf Command
The current behavior of the **resetpdf** command is the same as the **cleanpdf** command and is retained for compatibility with earlier releases on other platforms.

showpdf Command
A command to display the call and block counts for all procedures executed in a profile-directed feedback training run (compilation under the options **-qpdf1** and **-qshowpdf**).

gxlc and gxlc++ Utilities
Invocation methods that translate a GNU C or GNU C++ invocation command into a corresponding **xlc** or **xlC** command and invokes the XL C/C++ compiler. The purpose of these utilities is to minimize the number of changes to makefiles used for existing applications built with the GNU compilers and to facilitate the transition to XL C/C++.

Program optimization

XL C/C++ provides several compiler options that can help you control the optimization of your programs. With these options, you can:

- Select different levels of compiler optimizations.
- Control optimizations for loops, floating point, and other types of operations.
- Optimize a program for a particular class of machines or for a very specific machine configuration, depending on where the program will run.

Optimizing transformations can give your application better overall performance at run time. C and C++ provides a portfolio of optimizing transformations tailored to various supported hardware. These transformations can:

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

Significant performance improvements are possible with relatively little development effort because the compiler is capable of sophisticated program analysis and transformation. Moreover, XL C/C++ enables programming models, such as OpenMP, which allow you to write high-performance code.

If possible, you should test and debug your code without optimization before attempting to optimize it.

For more information about optimization techniques, see Optimizing your applications in the *XL C/C++ Advanced Edition V8.0 for Linux Programming Guide*.

For a summary of optimization-related compiler options, see Options for performance optimization in the *XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference*.

64-bit object capability

The XL C/C++ compiler's 64-bit object capability addresses increasing demand for larger storage requirements and greater processing power. The Linux operating system provides an environment that allows you to develop and execute programs that exploit 64-bit processors through the use of 64-bit address spaces.

To support larger executables that can be fit within a 64-bit address space, a separate, 64-bit object form is used to meet the requirements of 64-bit executables. The linker binds 64-bit objects to create 64-bit executables. Note that objects that are bound together must all be of the same object format. The following scenarios are not permitted and will fail to load, or execute, or both:

- A 64-bit object or executable that has references to symbols from a 32-bit library or shared library
- A 32-bit object or executable that has references to symbols from a 64-bit library or shared library
- A 64-bit executable that explicitly attempts to load a 32-bit module
- A 32-bit executable that explicitly attempts to load a 64-bit module
- Attempts to run 64-bit applications on 32-bit platforms

On both 64-bit and 32-bit platforms, 32-bit executables will continue to run as they currently do on a 32-bit platform.

XL C/C++ supports 64-bit mode mainly through the use of the **-q64** and **-qarch** compiler options. This combination determines the bit mode and instruction set for the target architecture.

For more information, see Using 32-bit and 64-bit modes in the *XL C/C++ Advanced Edition V8.0 for Linux Programming Guide*.

Shared memory parallelization

XL C/C++ Advanced Edition V8.0 for Linux supports application development for multiprocessor system architectures. You can use any of the following methods to develop your parallelized applications with XL C/C++:

- Directive-based shared memory parallelization (OpenMP)
- Instructing the compiler to automatically generate shared memory parallelization
- Message-passing-based shared or distributed memory parallelization (MPI)
- POSIX threads (Pthreads) parallelization
- Low-level UNIX parallelization using `fork()` and `exec()`

For more information, see Parallelizing your programs in the *XL C/C++ Advanced Edition V8.0 for Linux Programming Guide*.

OpenMP directives

OpenMP directives are a set of API-based commands supported by XL C/C++ and many other IBM and non-IBM C, C++, and Fortran compilers.

You can use OpenMP directives to instruct the compiler how to parallelize a particular loop. The existence of the directives in the source removes the need for the compiler to perform any parallel analysis on the parallel code. OpenMP directives require the presence of Pthread libraries to provide the necessary infrastructure for parallelization.

OpenMP directives address three important issues of parallelizing an application:

1. Clauses and directives are available for scoping variables. Frequently, variables should not be shared; that is, each processor should have its own copy of the variable.
2. Work sharing directives specify how the work contained in a parallel region of code should be distributed across the SMP processors.
3. Directives are available to control synchronization between the processors.

XL C/C++ supports the OpenMP API Version 2.5 specification. For more information, see www.openmp.org.

Diagnostic listings

The compiler output listing has optional sections that you can include or omit. For more information about the applicable compiler options and the listing itself, refer to "XL C/C++ compiler listings" on page 29.

Symbolic debugger support

You can use **gdb** or any other symbolic debugger when debugging your programs.

Chapter 2. What's new for V8.0

The new features and enhancements in XL C/C++ Advanced Edition V8.0 for Linux fall into four categories:

- "Performance and optimization"
- "Support for language enhancements and APIs" on page 13
- "Ease of use" on page 13
- "New compiler options" on page 14

Performance and optimization

Many new features and enhancements fall into the category of optimization and performance tuning.

Architecture and processor-specific code tuning

The **-qarch** compiler option controls the particular instructions that are generated for the specified machine architecture. The **-qtune** compiler option adjusts the instructions, scheduling, and other optimizations to enhance performance on the specified hardware. These options work together to generate application code that gives the best performance for the specified architecture.

XL C/C++ V8.0 augments the list of suboptions available to the **-qarch** compiler option to support processors that support the VMX instruction set and the newly-available POWER5+ processors. The following new **-qarch** options are available:

- -qarch=ppc64v
- -qarch=pwr5x

High performance libraries

XL C/C++ includes highly-tuned mathematical functions that can greatly improve the performance of mathematically-intensive applications. These functions are provided through the following high-performance libraries:

Mathematical Acceleration Subsystem (MASS)

MASS libraries provide high-performance scalar and vector functions to perform common mathematical computations. The MASS libraries included with XL C/C++ Advanced Edition V8.0 for Linux introduce new scalar and vector functions, and new support for the POWER5 processor architecture.

For more information about using the MASS libraries, see Using the Mathematical Acceleration Subsystem in the *XL C/C++ Advanced Edition V8.0 for Linux Programming Guide*.

Basic Linear Algebra Subprograms (BLAS)

XL C/C++ Advanced Edition V8.0 for Linux introduces the BLAS set of high-performance algebraic functions. You can use these functions to:

- Compute the matrix-vector product for a general matrix or its transpose.
- Perform combined matrix multiplication and addition for general matrices or their transposes.

For more information about using the BLAS functions, see Using the Basic Linear Algebra Subprograms in the *XL C/C++ Advanced Edition V8.0 for Linux Programming Guide*.

Other performance-related compiler options and directives

The entries in the following table describes new or changed compiler options and directives not already mentioned in the sections above.

Information presented here is just a brief overview. For more information about these compiler options, refer to Options for performance optimization in the *XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference*.

Table 3. Other Performance-Related Compiler Options and Directives

Option/directive	Description
-qfloat	**-qfloat** adds the following new suboptions: **-qfloat=relax** This suboption relaxes strict-IEEE conformance in exchange for greater speed, typically by removing trivial floating-point arithmetic operations such as adds and subtracts involving a zero on the right. **-qfloat=norelax** This is the default. Strict IEEE conformance is maintained.
-qipa	**-qipa** adds the following new suboptions: **-qipa=clonearch=***arch***{,***arch***}** Specifies one or more processor architectures for which multiple versions of the same instruction set are produced. XL C/C++ lets you specify multiple specific processor architectures for which instruction sets will be generated. At run time, the application will detect the specific architecture of the operating environment and select the instruction set specialized for that architecture. **-qipa=cloneproc=***name***{,***name***}** Specifies the names of one or more functions to clone for the processor architectures specified by the **clonearch** suboption.
-O	Specifying the **-O3** compiler option now instructs the compiler to also assume the **-qhot=level=0** compiler option setting. Specifying the **-O4** or **-O5** compiler option now instructs the compiler to also assume the **-qhot=level=1** compiler option setting.

Built-in functions new for this release

The following table lists built-in functions that are new for this release. For more information on built-in functions provided by XL C/C++, see Built-in functions for POWER and PowerPC architectures in the *XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference*.

Table 4. Built-in functions for XL C/C++

Function	Description
`void __builtin_return_address (unsigned int` *`level`*`);`	Returns the return address of the current function, or of one of its callers where *level* is a constant literal indicating the number of frames to scan up the call stack.
`void __builtin_frame_address (unsigned int` *`level`*`);`	Returns the address of the function frame of the current function, or of one of its callers where *level* is a constant literal indicating the number of frames to scan up the call stack
`int __compare_and_swap(volatile int*` *`addr`*`, int*` *`old_val_addr`*`, int` *`new_val`*`);`	Performs an atomic operation which compares the contents of a single word variable with a stored old value.
`int __compare_and_swaplp(volatile long*` *`addr`*`, long*` *`old_val_addr`*`, long` *`new_val`*`);`	Performs an atomic operation which compares the contents of a double word variable with a stored old value.
`int __fetch_and_add(volatile int*` *`addr`*`, int` *`val`*`);`	Increments the single word specified by *addr* by the amount specified by *val* in a single atomic operation.
`long __fetch_and_addlp(volatile long*` *`addr`*`, long` *`val`*`);`	Increments the double word specified by *addr* by the amount specified by *val* in a single atomic operation.
`unsigned int __fetch_and_and(volatile unsigned int*` *`addr`*`, unsigned int` *`val`*`);`	Clears bits in the single word specified by *addr* by AND-ing that value with the input *val* parameter, in a single atomic operation.
`unsigned long __fetch_and_andlp(volatile unsigned long*` *`addr`*`, unsigned long` *`val`*`);`	Clears bits in the double word specified by *addr* by AND-ing that value with the input *val* parameter, in a single atomic operation.
`unsigned int __fetch_and_or(volatile unsigned int*` *`addr`*`, unsigned int` *`val`*`);`	Sets bits in the single word specified by *addr* by OR-ing that value with the input *val* parameter, in a single atomic operation.
`unsigned long __fetch_and_orlp(volatile unsigned long*` *`addr`*`, unsigned long` *`val`*`);`	Sets bits in the double word specified by *addr* by OR-ing that value with the input *val* parameter, in a single atomic operation.
`unsigned int __fetch_and_swap(volatile unsigned int*` *`addr`*`, unsigned int` *`val`*`);`	Sets the single word specified by *addr* to the value or the input *val* parameter and returns the original contents of the memory location, in a single atomic operation.
`double __frim(double` *`val`*`);`	Takes an input *val* in double format, rounds *val* down to the next lower integral value, and returns the result in double format. Valid only for POWER5+ processors.

Table 4. Built-in functions for XL C/C++ (continued)

Function	Description
`float __frims(float val);`	Takes an input *val* in float format, rounds *val* down to the next lower integral value, and returns the result in float format. Valid only for POWER5+ processors.
`double __frin(double val);`	Takes an input *val* in double format, rounds *val* to the nearest integral value, and returns the result in double format. Valid only for POWER5+ processors.
`float __frins(float val);`	Takes an input *val* in float format, rounds *val* to the nearest integral value, and returns the result in float format. Valid only for POWER5+ processors.
`double __frip(double val);`	Takes an input *val* in double format, rounds *val* up to the next higher integral value, and returns the result in double format. Valid only for POWER5+ processors.
`float __frips(float val);`	Takes an input *val* in float format, rounds *val* up to the next higher integral value, and returns the result in float format. Valid only for POWER5+ processors.
`double __friz(double val);`	Takes an input *val* in double format, rounds *val* to the next integral value closest to zero, and returns the result in double format. Valid only for POWER5+ processors.
`float __frizs(float val);`	Takes an input *val* in float format, rounds *val* to the next integral value closest to zero, and returns the result in float format. Valid only for POWER5+ processors.
`long __ldarx(volatile long* addr);`	Generates a Load Double Word And Reserve Indexed (ldarx) instruction. This instruction can be used in conjunction with a subsequent stwcx. instruction to implement a read-modify-write on a specified memory location.
`int __lwarx(volatile int* addr);`	Generates a Load Word And Reserve Indexed (lwarx) instruction. This instruction can be used in conjunction with a subsequent stwcx. instruction to implement a read-modify-write on a specified memory location.
`int __stdcx(volatile long* addr, long val);`	Generates a Store Double Word Conditional Indexed (stdcx.) instruction. This instruction can be used in conjunction with a preceding ldarx instruction to implement a read-modify-write on a specified memory location.

Table 4. Built-in functions for XL C/C++ (continued)

Function	Description
`int __stwcx(volatile int* addr, int val);`	Generates a Store Word Conditional Indexed (stwcx.) instruction. This instruction can be used in conjunction with a preceding lwarx instruction to implement a read-modify-write on a specified memory location.
`unsigned long __mftb();`	Generates a Move From Time Base (mftb) hardware instruction.
`unsigned int __mftbu();`	Generates a Move From Time Base Upper (mftbu) hardware instruction.

Support for language enhancements and APIs

API and language enhancements can offer you additional ease of use and flexibility when developing your applications, as well as making it easier for you to develop code that more fully exploits the capabilities of your hardware platform.

OpenMP API V2.5 support for C, C++, and Fortran

XL C/C++ now supports the OpenMP API V2.5 standard. This latest level of the OpenMP specification combines the previous C/C++ and Fortran OpenMP specifications into one single specification for both C/C++ and Fortran, and resolves previous inconsistencies between them.

The OpenMP Application Program Interface (API) is a portable, scalable programming model that provides a standard interface for developing user-directed shared-memory parallelization in C, C++, and Fortran applications. The specification is defined by the OpenMP organization, a group of computer hardware and software vendors, including IBM.

You can find more information about OpenMP specifications at:

www.openmp.org

Ease of use

XL C/C++ includes the following new features to help you more easily use the compiler for your application development.

New installation and configuration utilities

This release of XL C/C++ introduces the **xlc_install** and **new_install** utilities to help you easily install and configure the compiler for initial use on your system.

Support for IBM Tivoli License Manager

IBM Tivoli License Manager (ITLM) is a Web-based solution that can help you manage software usage metering and license allocation services on supported systems. In general, ITLM recognizes and monitors the products that are installed and in use on your system.

IBM XL C/C++ Advanced Edition V8.0 for Linux is ITLM-enabled for inventory and usage signature support, which means that ITLM is able to detect both product installation of XL C/C++ and its usage.

Note: ITLM is not a part of the XL C/C++ compiler offering, and must be purchased and installed separately.

Once installed and activated, ITLM scans your system for product inventory signatures that indicate whether a given product is installed on your system. ITLM also identifies that product's version, release, and modification levels. Signature files for XL C/C++ are installed to the following directory:

Default installations
`/opt/ibmcmp/vac/8.0`

Non-default installations
`compiler``/vac/8.0` where *`compiler`* is the target directory for installation specified by the **--prefix** installation option.

For more information about IBM Tivoli License Manager Web, see:

www.ibm.com/software/tivoli/products/license-mgr

New compiler options

Compiler options can be specified on the command line or through directives embedded in your application source files.

New command line options

The following table summarizes command line options new to XL C/C++. You can find detailed syntax and usage information for all compiler options in Compiler options reference in the *XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference*.

Option	Description and remarks
-qasm	The **-qasm** compiler option now adds new functionality. You can now not only use this compiler option to control how inline assembler statements in your program are interpreted, but you can also control whether or not code is emitted for the asm statement.
-qasm_as	The syntax of the **-qasm_as** compiler option has changed slightly.
-qdump_class_hierarchy	When this option is in effect, the compiler dumps a representation of the hierarchy and virtual function table layout for each class object to a file.
-qlist	The **-qlist** compiler option adds new **offset** and **nooffset** suboptions. Specifying **-qlist=offset** instructs the compiler to show object listing offsets from the start of a procedure rather than from the start of code generation.
-qmakedep	The **-qmakedep** compiler option adds a new **gcc** suboption. Specifying **-qmakedep=gcc** instructs the compiler to generate make dependency information in a format similar to that used by the GNU C/C++ compiler.
-MF	This new compiler option specifies a filename for the make dependency file generated by the **-qmakedep** or **-M** option.

Option	Description and remarks
-qppline	This new compiler option enables generation of #line directives in preprocessed output. The **-qnoppline** compiler option disables generation of #line directives.
-qreserved_reg	This new compiler option lets you reserve one or more register names. A reserved register cannot be used during compilation except as a stack pointer, frame pointer or in some other fixed role.
-qsourcetype	This release adds **assembler-with-cpp** as a new suboption to the **-qsourcetype** compiler option. Ordinarily, the compiler recognizes assembler source files that require preprocessing by the file's **.S** filename suffix. The compiler preprocesses **.S** source files and then sends the preprocessor output to the assembler. Specifying **-qsourcetype=assembler-with-cpp** *filename* on the command line instructs the compiler to treat all filenames appearing after the **assembler-with-cpp**, regardless of filename suffix, as being assembler source files requiring preprocessing.
-qtmplinst	This new compiler option manages how the compiler performs implicit instantiations of templates.
-qversion	Specifying the **-qversion** compiler option returns the official compiler product name and version.

New pragma directives

The following table summarizes pragma directive options new to XL C/C++. You can find detailed syntax and usage information in XL C/C++ Pragmas in the *XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference*.

#pragma Directive	Description and remarks
`altivec_vrsave`	When the `altivec_vrsave` directive is in effect, function prologs and epilogs include code to maintain the VRSAVE register. This pragma has effect only when **-qaltivec** is in effect, must be used only within a function, and affects only the function in which it appears.
`STDC CX_LIMITED_RANGE`	The `STDC CX_LIMITED_RANGE` directive instructs the compiler that within the scope it controls, complex division and absolute value are only invoked with values such that intermediate calculation will not overflow or lose significance.

Chapter 3. Setting up and customizing XL C/C++

This section provides brief overview information about setting up and customizing XL C/C++, together with pointers to other documentation that describes specific set-up and customization topics in greater detail.

Environment variables and XL C/C++

XL C/C++ uses a number of environment variables to control various aspects of compiler operation. Environment variables fall into two basic categories:

- Environment variables defining the basic working environment for the compiler.
- Environment variables defining run time compiler option defaults.

Setting the compiler working environment

These environment variables define the basic working environment for the compiler, including specifying your choice of national language or defining the location of libraries or temporary files. For complete information, refer to Setting up the compilation environment in the *XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference*.

LANG
Specifies the default national language *locale* used to display diagnostic messages and compiler listings. This environment variable also affects runtime behavior.

MANPATH
Specifies the search path for system, compiler, and third-party man pages.

NLSPATH
Specifies one or more directory locations where message catalogs can be found. This environment variable also affects runtime behavior.

PDFDIR
Specifies the directory location where profile-directed feedback information is stored when you compile with the **-qpdf** option.

TMPDIR
Specifies the directory location where the compiler will store temporary files created during program compilation. This environment variable also affects runtime behavior.

Setting the default runtime options

These environment variables define runtime compiler option defaults to be used by the compiler, unless explicitly overridden by compiler option settings specified on the command line or in directives located in your program source. For complete information, refer to Setting up the compilation environment in the *XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference*.

XL_NOCLONEARCH
Instructs the program to execute only generic code, where generic code is compiled object code that is not versioned for a specific processor architecture. You can set the XL_NOCLONEARCH environment variable to help you debug your application.

XLSMPOPTS
The **XLSMPOPTS** environment variable allows you to specify run time options that affect SMP execution.

OMP_DYNAMIC, OMP_NESTED, OMP_NUM_THREADS, OMP_SCHEDULE
These environment variables, are part of the OpenMP standard. They let you specify how the application will execute sections of parallel code.

Customizing the configuration file

The configuration file is a plain text file that specifies default settings for compiler options and invocations. XL C/C++ provides a default configuration at file /etc/opt/ibmcmp/vac/8.0/vac.cfg during compiler installation.

If you are running on a single-user system, or if you already have a compilation environment with compilation scripts or makefiles, you may want to leave the default configuration file as it is.

As an alternative, you can create additional custom configuration files to meet special compilation requirements demanded by specific applications or groups of applications.

See Customizing the configuration file in the *XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference* for more information on creating and using custom configuration files.

Determining what level of XL C/C++ is installed

You may not be sure which level of XL C/C++ is installed on a particular machine. You will need to know this information if contacting software support.

To display the version and PTF release level of the compiler you have installed on your system, invoke the compiler with the **-qversion** compiler option. For example:

```
xlC -qversion
```

Chapter 4. Editing, compiling, and linking programs with XL C/C++

Basic C and C++ program development consists of repeating cycles of editing, compiling and linking (by default a single step combined with compiling), and running.

Prerequisite Information:

1. Before you can use the compiler, you must first ensure that all Linux settings (for example, certain environment variables and storage limits) are correctly configured. For more information see "Environment variables and XL C/C++" on page 17.
2. To learn more about writing C and C++ programs, refer to the *XL C/C++ Advanced Edition V8.0 for Linux Language Reference*.

The compiler phases

The typical compiler invocation command executes some or all of the following programs in sequence. For link time optimizations, some of the phases will be executed more than once during a compilation. As each program runs, the results are sent to the next step in the sequence.

1. A preprocessor
2. The compiler, which consists of the following phases:
   a. Front-end parsing and semantic analysis
   b. Loop transformations
   c. Interprocedural analysis
   d. Optimization
   e. Register allocation
   f. Final assembly
3. The assembler (for **.s** files and for **.S** files after they are preprocessed)
4. The linker **ld**

To see the compiler step through these phases, specify the **-qphsinfo** and **-v** compiler options when you compile your application.

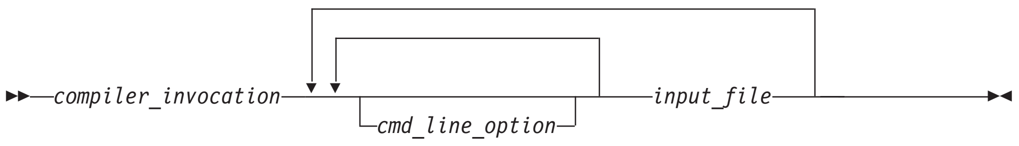
Editing C and C++ source files

To create C and C++ source programs, you can use any of the available text editors, such as **vi** or **emacs**. Source programs must use a recognized filename suffix unless the configuration file or the **-qsourcetype** compiler option define additional non-standard filename suffixes. See "XL C/C++ input files" on page 21 for a list of filename suffixes recognized by XL C/C++.

For a C or C++ source program to be a valid program, it must conform to the language definitions specified in the *XL C/C++ Advanced Edition V8.0 for Linux Language Reference*.

Compiling with XL C/C++

To compile a source program, use one of the compiler invocation commands with the syntax shown below:



The compiler invocation command performs all necessary steps to compile C or C++ source files, assemble any **.s** and **.S** files, and link the object files and libraries into an executable program.

For new C or C++ application work, you should consider compiling with xlC or xlc++, or its threadsafe counterparts.

Both **xlC** and **xlc++** will compile program source as either C or C++, but compiling C++ files with **xlc** may result in link or runtime errors because libraries required for C++ code are not specified when the linker is called by the C compiler. The other base compiler invocation commands exist primarily to provide explicit compilation support for different levels and extensions of the C or C++ language.

In addition to the base compiler invocation commands, XL C/C++ also provides specialized variants of many base compiler invocations. A variation on a base compiler invocation is named by attaching a suffix to the name of that invocation command. Suffix meanings for invocation variants are:

_r	Threadsafe invocation variant that supports POSIX Pthread APIs for multithreaded applications, including applications compiled with **-qsmp** or with source code containing OpenMP program parallelization directives.

Table 5. XL C/C++ compiler invocation commands

Base Invocation	Variants on Base Invocation	Description
xlC xlc++	xlC_r xlc++_r	Invokes the compiler so that source files are compiled as C++ language source code.
xlc	xlc_r	Invokes the compiler so that source files are compiled as C source code.
c99	c99_r	Invokes the compiler so that source files are compiled with strict conformance to the ISO C99 standard (ISO/IEC 14882:1999). **Note:** The ISO C99 standard also specifies features in the runtime library. These features may not be supported in the runtime library currently installed on your system.
c89	c89_r	Invokes the compiler so that source files are compiled with strict conformance to the ISO C89 standard (ISO/IEC 9899:1990).
cc	cc_r	Invokes the compiler for use with legacy C code that does not require compliance with C89 or C99.

Table 5. XL C/C++ compiler invocation commands (continued)

Base Invocation	Variants on Base Invocation	Description
gxlc		Invokes the compiler after translating GNU C command-line options to XL C/C++ options. **Note:** Not every GNU C option has an exact XL C/C++ equivalent.
gxlC gxlc++		Invokes the compiler after translating GNU C++ command-line options to XL C/C++ options. **Note:** Not every GNU C++ option has an exact XL C/C++ equivalent.

Compiling parallelized XL C/C++ applications

XL C/C++ provides threadsafe compilation invocations that you can use when compiling parallelized applications for use in multiprocessor environments.

- xlC_r
- xlc++_r
- xlc_r
- c99_r
- c89_r
- cc_r

These invocations are similar to their corresponding base compiler invocations, except that they link and bind compiled objects to threadsafe components and libraries.

Note: Using any of these commands alone does not imply parallelization. For the compiler to recognize OpenMP directives and activate parallelization, you must also specify **-qsmp** compiler option. In turn, you can only specify the **-qsmp** option in conjunction with one of these six invocation commands. When you specify **-qsmp**, the driver links in the libraries specified on the smp libraries line in the active stanza of the configuration file.

XL C/C++ input files

The input files to the compiler are:

Source files (.c suffix for C language, .C .cc .cp .cpp .cxx .c++ suffixes for C++ language)

The compiler considers files with these suffixes as being C or C++ source files for compilation.

The compiler compiles source files in the order you specify on the command line. If it cannot find a specified source file, the compiler produces an error message and proceeds to the next file, if one exists.

If you have C or C++ source files that do not conform to standard C or C++ file naming conventions, you can use the **-+** compiler option to instruct the compiler to treat such files as C or C++ source files. Such files, other than those with .a, .o, .so, .s, or .S filename suffixes, are compiled as C++ source files when the **-+** compiler option is in effect.

Include files also contain source and often have suffixes different from those ordinarily used for C or C++ source files.

Preprocessed source files (.i suffix)

The compiler sends the preprocessed source file, `filename.i`, to the compiler where it is preprocessed again in the same way as a `.c` or `.C` file. Preprocessed files are useful for checking macros and preprocessor directives.

Object files (.o suffix)

After the compiler compiles the source files, it uses the **ld** command to link the resulting .o files, any .o files that you specify as input files, and some of the .o and .a files in the product and system library directories. The compiler can then produce a single .o object file or a single executable output file from these object files.

Assembler source files (.s and .S suffixes)

The compiler sends assembler source files to the assembler (**as**). The assembler sends object files to the linker at link time.

Note: Assembler source files with a .S filename suffix are first preprocessed by the compiler, then sent to the assembler.

Shared object or library files (.so suffix)

These are object files that can be loaded and shared by multiple processes at run time. When a shared object is specified during linking, information about the object is recorded in the output file, but no code from the shared object is actually included in the output file.

Configuration files (.cfg suffix)

The contents of the configuration file determine many aspects of the compilation process, most commonly the default options for the compiler. You can use it to centralize different sets of default compiler options or to keep multiple levels of the XL C/C++ compiler present on a system.

The default configuration files are /etc/opt/ibmcmp/vac/8.0/vac.cfg and /etc/opt/ibmcmp/vac/8.0/gxlc.cfg.

Profile data files

The **-qpdf1** option produces runtime profile information for use in subsequent compilations. This information is stored in one or more hidden files with names that match the pattern .*pdf*.

XL C/C++ output files

The output files that C and C++ produces are:

Executable files: a.out

By default, XL C/C++ produces an executable file that is named **a.out** in the current directory.

Object files: ***filename*.o**

If you specify the **-c** compiler option, instead of producing an executable file, the compiler produces an object file for each specified program source input file, and the assembler produces an object file for each specified assembler input file. By default, the output object files are saved to the current directory using the same file name prefixes as their corresponding source input files.

Assembler source files: ***filename*.s**

If you specify the **-S** compiler option, instead of producing an executable file, the XL C/C++ compiler produces an equivalent assembler source file for each specified input source file. By default, the output assembler source files are saved to the current directory using the same file name prefixes as their corresponding source input files.

Compiler listing files: *filename*.**lst**
: By default, no listing is produced unless you specify one or more listing-related compiler options. The listing file is placed in the current directory, with the same file name prefix as the source file.

cpp-Preprocessed source files: *filename*.**i**
: To create a preprocessed source file, specify the **-P** option at compilation time. The source files are preprocessed but not compiled. You can also redirect the output from the **-E** option to generate a preprocessed file that contains #line directives. A preprocessed source file, `filename.i`, is produced for each source file. By default, output preprocessor source files are saved to the current directory using the same file name prefixes as their corresponding source input files.

Make dependency files: *filename*.**u**
: When the **-M** or **-qmakedep** compiler option is in effect, the compiler creates a .u file for each C or C++ source file compiled. You can use the dependency information provided by .u files to help you create make files.

Each .u file contains a line for the input file and an entry for each include file in the general form of:

```
file_name.o :file_name.c
file_name.o :include_file_name
```

Include files are listed according to the compiler's search order rules for the `#include` preprocessor directive. If the include file is not found, it is not added to the .u file. Files with no include statements produce output files containing one line that lists only the input file name.

Profile data files (.*pdf*)
: These are the profile-directed feedback files that the **-qpdf1** compiler option produces. They are used in subsequent compilations to tune optimizations according to actual execution results.

Specifying compiler options

Compiler options perform a variety of functions, such as setting compiler characteristics, describing the object code to be produced, controlling the diagnostic messages emitted, and performing some preprocessor functions.

You can specify compiler options:

- On the command line with command line compiler options.
- In the stanzas found in a compiler configuration file
- In your source code using directive statements
- Or by using any combination of these techniques.

When multiple compiler options have been specified, it is possible for option conflicts and incompatibilities to occur. To resolve these conflicts in a consistent fashion, the compiler usually applies the following general priority sequence:

1. Directive statements in your source file *override* command line settings
2. Command line compiler option settings *override* configuration file settings
3. Configuration file settings *override* default settings

Generally, if the same compiler option is specified more than once on a command line when invoking the compiler, the last option specified prevails.

Note: The **-I** compiler option is a special case. The compiler searches any directories specified with **-I** in the **vac.cfg** file before it searches the directories specified with **-I** on the command line. The option is cumulative rather than preemptive.

Other options with cumulative behavior are **-R** and **-l** (lowercase L).

You can also pass compiler options to the linker, assembler, and preprocessor. See Compiler options reference in the *XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference* for more information about compiler options and how to specify them.

Linking XL C/C++ programs

By default, you do not need to do anything special to link an XL C/C++ program. The compiler invocation commands automatically call the linker to produce an executable output file. For example, running the following command:

```
xlC file1.C file2.o file3.C
```

compiles and produces the object files `file1.o` and `file3.o`, then all object files (including `file2.o`) are submitted to the linker to produce one executable.

After linking, follow the instructions in Chapter 5, "Running XL C/C++ programs," on page 27 to execute the program.

Compiling and linking in separate steps

To produce object files that can be linked later, use the **-c** option.

```
xlc++ -c file1.C              # Produce one object file (file1.o)
xlc++ -c file2.C file3.C      # Or multiple object files (file1.o, file3.o)
xlc++ file1.o file2.o file3.o  # Link object files with appropriate libraries
```

It is often best to execute the linker through the compiler invocation command, because it passes some extra **ld** options and library names to the linker automatically.

Dynamic and static linking

XL C/C++ allows your programs to take advantage of the operating system facilities for both dynamic and static linking:

- Dynamic linking means that the code for some external routines is located and loaded when the program is first run. When you compile a program that uses shared libraries, the shared libraries are dynamically linked to your program by default.

  Dynamically linked programs take up less disk space and less virtual memory if more than one program uses the routines in the shared libraries. During linking, they do not require any special precautions to avoid naming conflicts with library routines. They may perform better than statically linked programs if several programs use the same shared routines at the same time. They also allow you to upgrade the routines in the shared libraries without relinking.

  Because this form of linking is the default, you need no additional options to turn it on.

- Static linking means that the code for all routines called by your program becomes part of the executable file.

  Statically linked programs can be moved to and run on systems without the XL C/C++ libraries. They may perform better than dynamically linked programs if they make many calls to library routines or call many small routines. They do require some precautions in choosing names for data objects and routines in the program if you want to avoid naming conflicts with library routines. They also may not work if you compile them on one level of the operating system and run them on a different level of the operating system.

See Invoking the linkage editor in the *XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference* for more information about linking your programs.

Also, see Constructing a library in the XL C/C++ Advanced Edition V8.0 for Linux Programming Guide for more information about compiling and linking a library.

Chapter 5. Running XL C/C++ programs

The default file name for the program executable file produced by the XL C/C++ compiler is **a.out**. You can select a different name with the **-o** compiler option.

You can run a program by entering the name of a program executable file together with any runtime arguments on the command line.

You should avoid giving your program executable file the same name as system or shell commands, such as **test** or **cp**, as you could accidentally execute the wrong command. If you do decide to name your program executable file with the same name as a system or shell command, you should execute your program by specifying the path name to the directory in which your program executable file resides, such as **./test**.

Canceling execution

To suspend a running program, press the **Ctrl+Z** key while the program is in the foreground. Use the **fg** command to resume running.

To cancel a running program, press the **Ctrl+C** key while the program is in the foreground.

Setting runtime options

You can use environment variable settings to control certain runtime options and behaviors of applications created with the XL C/C++ compiler. Other environment variables do not control actual runtime behavior, but can have an impact on how your applications run.

For more information on environment variables and how they can affect your applications at runtime, see “Environment variables and XL C/C++” on page 17.

Running compiled applications on other systems

If you want to run an application developed with the XL C/C++ compiler on another system that does not have the compiler installed, you will need to install a runtime environment on that system.

You can obtain the latest XL C/C++ Runtime Environment install images, together with licensing and usage information, from the Download section on the XL C/C++ Support page at:

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

Chapter 6. XL C/C++ compiler diagnostic aids

XL C/C++ issues diagnostic messages when it encounters problems compiling your application. You can use these messages to help identify and correct such problems.

This section provides a brief overview of the main diagnostics messages offered by XL C/C++. For more information about related compiler options that can help you resolve problems with your application, see Options for error checking and debugging and Options that control listings and messages in the *XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference*.

Compilation return codes

At the end of compilation, the compiler sets the return code to zero under any of the following conditions:
- No messages are issued.
- The highest severity level of all errors diagnosed is less than the setting of the **-qhalt** compiler option, and the number of errors did not reach the limit set by the **-qmaxerr** compiler option.
- No message specified by the **-qhaltonmsg** compiler option is issued.

Otherwise, the compiler sets the return code to one of the following values:

Return Code	Error Type
1	An error with a severity level higher than the setting of the **-qhalt** compiler option has occurred.
40	An option error or unrecoverable error has occurred.
41	A configuration file error has occurred.
250	An out-of-memory has occurred. The compiler invocation command cannot allocate any more memory for its use.
251	A signal-received error has occurred. That is, an unrecoverable error or interrupt signal has occurred.
252	A file-not-found error has occurred.
253	An input/output error has occurred - files cannot be read or written to.
254	A fork error has occurred. A new process cannot be created.
255	An error has been detected while the process was running.

Note: Return codes may also be displayed for runtime errors.

XL C/C++ compiler listings

Diagnostic information is produced in the output listing according to the settings of the **-qlist**, **-qsource**, **-qxref**, **-qattr**, **-qreport**, and **-qlistopt** compiler options. The **-S** option generates an assembler listing in a separate file.

If the compiler encounters a programming error when compiling an application, the compiler issues a diagnostic message to the standard error device and, if the appropriate compiler options have been selected, to a listing file.

To locate the cause of a problem with the help of a listing, you can refer to:

- The source section (to see any compilation errors in the context of the source program)
- The attribute and cross-reference section (to find data objects that are misnamed or used without being declared or to find mismatched parameters)
- The transformation and object sections (to see if the generated code is similar to what you expect)

A heading identifies each major section of the listing. A string of "greater than" (>) symbols precede the section heading so that you can easily locate its beginning:

```
>>>>> section name
```

You can select which sections appear in the listing by specifying the appropriate compiler options. For more information about these options see Options for error checking and debugging and Options that control listings and messages in the *XL C/C++ Advanced Edition V8.0 for Linux Compiler Reference*.

Header section

The listing file has a header section that contains the following items:

- A compiler identifier that consists of the following:
  - Compiler name
  - Version number
  - Release number
  - Modification number
  - Fix number
- Source file name
- Date of compilation
- Time of compilation

The header section is always present in a listing; it is the first line and appears only once. The following sections are repeated for each compilation unit when more than one compilation unit is present.

Options section

The options section is always present in a listing. There is a separate section for each compilation unit. It indicates the specified options that are in effect for the compilation unit. This information is useful when you have conflicting options. If you specify the **-qlistopt** compiler option, this section lists the settings for all options.

Source section

The source section contains the input source lines with a line number and, optionally, a file number. The file number indicates the source file (or include file) from which the source line originated. All main file source lines (those that are not from an include file) do not have the file number printed. Each include file has a file number associated with it, and source lines from include files have that file number printed. The file number appears on the left, the line number appears to its right, and the text of the source line is to the right of the line number. The compiler numbers lines relative to each file. The source lines and the numbers that are associated with them appear only if the **-qsource** compiler option is in effect.

If the **-qsource** option is in effect, the error messages are interspersed with the source listing. The error messages that are generated during the compilation process contain the following:

- The source line
- A line of indicators that point to the columns that are in error
- The error message.

For example:

```
          7 |     for (i=0; i<100; i++) {
          8 |         for (j=0; j<100; j++) {
          9 |             a[i:j] = j;
"loop.c", line 9.16: 1506-277 (S) Syntax error: possible missing ',' or ']'?
         10 |         }
         11 |     }
```

If the **-qnosource** option is in effect, the error messages are all that appear in the source section, and an error message contains:

- The file name in quotation marks
- The line number and column position of the error
- The error message.

For example:

```
"loop.c", line 9.16: 1506-277 (S) Syntax error: possible missing ',' or ']'?
```

Error message format

The format of a C and C++ diagnostic message is:

```
►►──15──cc──-──nnn── ─┬──────────────────────┬─message_text─────────────►◄
                      └─(──severity_letter──)─┘
```

where:

15*cc nnn*
: Indicates an XL C/C++ message and component number.

severity_letter
: Indicates how serious the problem is, as described in the preceding section.

: Compilation errors can have the following severity levels, which are displayed as part of some error messages:

: **U** An unrecoverable error. Compilation failed because of an internal compiler error.

: **S** A severe error. Compilation failed due to one of the following:
  - Conditions exist that the compiler could not correct.
  - An internal compiler table has overflowed. Processing of the program stops, and XL C/C++ does not produce an object file.
  - An include file does not exist. Processing of the program stops, and XL C/C++ does not produce an object file.
  - An unrecoverable program error has been detected. Processing of the source file stops, and XL C/C++ does not produce an object file. You can usually correct this error by fixing any program errors that were reported during compilation.

E	C compilations only. The compiler detected an error in your source code and attempted to correct it. The compiler will continue to compile your application, but might not generate the results you expect.
W	Warning message. The compiler detected a potential problem in your source code, but did not attempt to correct it. The compiler will continue to compile your application, but might not generate the results you expect.
I	Informational message. It does not indicate any error, just something that you should be aware of to avoid unexpected behavior.

'*message text***'**	Is the text describing the error

By default, the compiler stops without producing output files if it encounters a severe error (severity S). You can make the compiler stop for less severe errors by specifying a different severity with the **-qhalt** option. For example, with **-qhalt=w**, the compiler stops if it encounters any errors of severity W or higher severity. This technique can reduce the amount of compilation time that is needed to check the syntactic and semantic validity of a program. You can limit low-severity messages without stopping the compiler by using the **-qflag** option.

Transformation report section

If the **-qreport** option is in effect, the compiler generates a transformation report showing how XL C/C++ optimized the program. This section displays pseudo-code that corresponds to the original source code, so that you can see parallelization and loop transformations generated by the **-qhot** or **-qsmp** options.

Attribute and cross-reference section

This section provides information about the entities that are used in the compilation unit. It is present if the **-qxref** or **-qattr** compiler option is in effect. Depending on the options in effect, this section contains all or part of the following information about the entities that are used in the compilation unit:

- Names of the entities
- Attributes of the entities (if **-qattr** is in effect). Attribute information may include any or all of the following details:
  - The data type
  - The class of the name
  - The relative address of the name
  - Alignment
  - Dimensions
  - For an array, whether it is allocatable
  - Whether it is a pointer, target, or integer pointer
  - Whether it is a parameter
  - Whether it is volatile
  - For a dummy argument, its intent, whether it is value, and whether it is optional
  - Private, public, protected, module
- Coordinates to indicate where you have defined, referenced, or modified the entities. If you declared the entity, the coordinates are marked with a `$`. If you initialized the entity, the coordinates are marked with a `*`. If you both declared and initialized the entity at the same place, the coordinates are marked with a `&`. If the entity is set, the coordinates are marked with a `@`. If the entity is referenced, the coordinates are not marked.

If you specify the **full** suboption with **-qxref** or **-qattr**, C and C++ reports all entities in the compilation unit. If you do not specify this suboption, only the entities you actually use appear.

Object section

XL C/C++ produces this section only when the **-qlist** compiler option is in effect. It contains a pseudo-assembler object code listing showing the source line number, the instruction offset in hexadecimal notation, the assembler mnemonic of the instruction, and the hexadecimal value of the instruction. On the right side, it also shows the cycle time of the instruction and the intermediate language of the compiler. Finally, the total cycle time (straight-line execution time) and the total number of machine instructions that are produced are displayed. There is a separate section for each compilation unit.

Note: To obtain a true assembler listing, specify the **-S** compiler option when compiling your application. The assembler listing will be named *filename*.s.

File table section

This section contains a table that shows the file number and file name for each main source file and include file used. It also lists the line number of the main source file at which the include file is referenced. This section is always present.

Compilation unit epilogue section

This is the last section of the listing for each compilation unit. It contains the diagnostics summary and indicates whether the unit was compiled successfully. This section is not present in the listing if the file contains only one compilation unit.

Compilation epilogue section

The compilation epilogue section occurs only once at the end of the listing. At completion of the compilation, XL C/C++ presents a summary of the compilation: number of source records that were read, compilation start time, compilation end time, total compilation time, total CPU time, and virtual CPU time. This section is always present in a listing.

Debugging compiled applications

Specifying the **-g** or **-qlinedebug** compiler options at compile time instructs the XL C/C++ compiler to include debugging information in compiled output. You can then use **gdb** or any other symbolic debugger to step through and inspect the behavior of your compiled application.

Notices

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

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

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

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

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

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

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

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

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

Programming interface information

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

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

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

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

Trademarks and service marks

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

AIX	IBM	OS/390
OS/400	POWER3	POWER4
POWER5	POWER5+	PowerPC
z/OS	z/VM	

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

OpenMP is a trademark of the OpenMP Architecture Review Board.

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

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

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

Index

Program Number: 5724-M16

SC09-8015-00