XL C/C++ Advanced Edition

Getting Started with IBM XL C/C++ Advanced Edition V7.0 for Linux

Version 7.0

XL C/C++ Advanced Edition

Getting Started with IBM XL C/C++ Advanced Edition V7.0 for Linux

Version 7.0

> **Note!**
>
> Before using this information and the product it supports, be sure to read the information in "Notices" on page 57.

Second Edition (March, 2005)

This edition applies to Version 7.0.1 of IBM XL C/C++ Advanced Edition (product number 5724-K77) and to all subsequent releases and modifications until otherwise indicated in new editions.

IBM welcomes your comments. You can send them by the Internet to the following address:

```
compinfo@ca.ibm.com
```

Include the title and order number of this book, and the page number or topic related to your comment. Be sure to include your e-mail address if you want a reply.

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 book

XL C/C++ Advanced Edition is an optimizing, standards-based, command-line compiler for the Linux operating system running on the PowerPC® architecture. The compiler is a professional programming tool for creating and maintaining 32-bit and 64-bit applications in the extended C and C++ programming languages.

This book introduces you to the XL C/C++ compiler. It describes the various compiler invocations and ways to customize the compilation environment and control the compilation process. This book contains descriptions of the types of transformations the compiler can perform, the accepted file types for input and output, categorized summaries of compiler options, and considerations for porting an existing application. This book also provides a brief introduction to optimizing the performance of your applications. The optimizing capabilities of the compiler enable you to exploit the multilayered architecture of the PowerPC processor.

Linux and AIX® are complementary operating systems. If you are familiar with IBM C for AIX or VisualAge® C++ Professional for AIX, your makefiles can be readily adapted to work on the Linux platform. If you are new to the IBM C and C++ compilers, this book can open a path to improved performance during compile, link, and run time.

This document assumes that you are familiar with the C and C++ programming languages, the Linux operating system, and the bash shell.

Highlighting conventions

Bold
: Identifies commands, keywords, and other items whose names are predefined by the system.

Italics
: Identify parameters whose actual names or values are to be supplied by the programmer. *Italics* are also used for the first mention of new terms.

`Example`
: Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code, messages from the system, or information that you should actually type.

Examples are intended to be instructional and do not attempt to minimize run time, conserve storage, or check for errors. The examples do not demonstrate all of the possible uses of the language constructs. Some examples are only code fragments and will not compile without additional code.

How to read the syntax diagrams

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

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

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

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

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

v

Reading the Syntax Diagrams

Diagrams of syntactical units other than complete commands, directives, or statements start with the ►—— symbol and end with the —→ symbol.

Note: In the following diagrams, `statement` represents a C or C++ command, directive, or statement.

- Required items appear on the horizontal line (the main path).

►►—statement—*required_item*————————————————————————►◄

- Optional items appear below the main path.

►►—statement———————————————————————————————————————►◄
         └─*optional_item*─┘
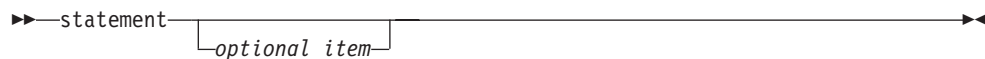
- If you can choose from two or more items, they appear vertically, in a stack.

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

►►—statement——┬─*required_choice1*─┬———————————————————►◄
            └─*required_choice2*─┘
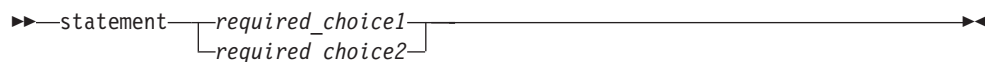
  If choosing one of the items is optional, the entire stack appears below the main path.

►►—statement——————————————————————————————————————►◄
           ├─*optional_choice1*─┤
           └─*optional_choice2*─┘

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

           ┌─*default_item*──┐
►►—statement——┴─*alternate_item*─┴——————————————————►◄

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

            ┌──────────────┐
►►—statement——▼─*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 appear in nonitalic letters and should be entered exactly as shown (for example, `extern`).

  Variables appear 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. See *XL C/C++ Language Reference* for information on the **#pragma** directive.

```
 1   2   3    4        5          6                          9    10
►►─#──pragma──comment──(──────────compiler───────────────────)─►◄
                          │────────date──────────────────────│
                          │────────timestamp─────────────────│
                          │────────copyright────────┐
                          └────────user─────────┐   └──,─"characters"──┘
                                                       7          8
```
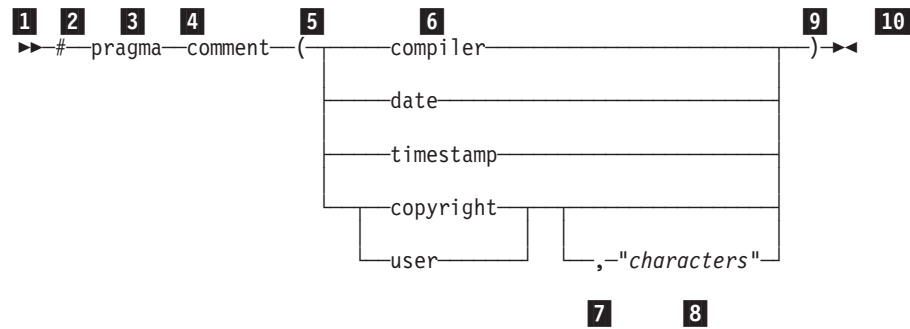
1 This is the start of the syntax diagram.

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")
```

Reading the Syntax Diagrams

XL C/C++ overview

XL C/C++ Advanced Edition is an optimizing, standards-based, command-line compiler for the Linux operating system running on PowerPC hardware with the PowerPC architecture. The compiler enables application developers to create and maintain optimized 32-bit and 64-bit applications. These applications can use automatic or explicit shared memory program parallelization to improve performance. Many of the optimizations and performance improvements made possible by the PowerPC architecture are controlled by compiler options, pragmas, and compiler invocation modes. The benefits of the hardware architecture can therefore be realized with a minimum amount of specialized hand coding.

This product is the follow-on release to IBM VisualAge C++ Version 6.0 for Linux. IBM has rebranded *VisualAge C++* as *XL C/C++*.

Shared memory program parallelization

Significant gains in program performance lie in exploiting the shared memory program parallelization (SMP) that is possible on the PowerPC architecture. The compiler offers the following methods of implementing SMP:
* Automatic parallelization of program loops (implicit parallelization).
* Explicit parallelization of C and C++ program code by using OpenMP-compliant pragma directives.

OpenMP directives involve defining parallel regions of iterative and non-iterative code.

Source compatibility

The compiler supports the two ISO programming language specifications for C: ISO/IEC 9899:1990 (referred to as C89) and ISO/IEC 9899:1999 (C99). The compiler also supports both of the C++ standards: ISO/IEC 14882:2003 (referred to as *Standard C++*) and ISO/IEC 14882:1998, the first official specification of the language (referred to as C++98). In addition to the standardized language levels, the compilers support numerous language extensions, which include a subset of the GNU C and C++ language extensions.

Binary compatibility

XL C/C++ supports the C++ Abstract Binary Interface (ABI). The compiler produces binary or object files that are compatible with those created with the GNU C/C++ Version 3.2 or 3.3 compiler. The compiler also supports the OpenMP version 2.0 standard. OpenMP-compliant C and C++ object files are binary compatible with those created by IBM XL Fortran for Linux, thus allowing interlanguage calls between C or C++ and Fortran.

Coexistence with open source resources

To achieve binary compatibility with code compiled with a GNU C or C++ compiler, a program compiled with XL C/C++ includes the same headers as those used by a GNU compiler residing on the same system. The compiler optimizes the program while maintaining binary compatibility with objects produced by GNU C. Some noteworthy points about this relationship are:
* IBM built-in functions for PowerPC processors coexist with GNU C built-ins.
* Compilation uses the GNU C and C++ header files from the Linux distribution.

- Compilation uses the GNU assembler.
- Linking uses the GNU linker.
- The compiled program uses the GNU C and C++ run-time libraries.
- Debugging uses the GNU debugger, gdb.

Command-line C and C++ compiler

XL C/C++ provides a selection of base compiler invocation commands, which support various version levels of the C and C++ languages. Each invocation command automatically sets a compiler suboption for language level, options for other related language features, and any related predefined macros. In most cases, you should use the **xlc** command to compile C source files and the **xlC** command to compile C++ source files, or when you have both C and C++ source files. The invocation **xlc++** is equivalent to **xlC**.

Variations of the base command are provided to support the requirements of special environments and file systems. A variation is formed by attaching a suffix to the base command. On the Linux platform, the _r suffix is provided for compiling thread-safe applications. These variations are also referred to as *reentrant invocations*.

In addition, the gxlc and gxlc++ utilities are specialized compiler invocations.

Libraries

XL C/C++ uses the GNU C and C++ headers, and the resulting application is linked with the C and C++ run-time libraries provided with the gcc level that is loaded with the operating system.XL C/C++ ships the SMP run-time library to support the automatic parallelization and OpenMP features of the XL C/C++ compilers.

High-performance mathematics libraries

Starting in Version 7, XL C/C++ ships the IBM Mathematics Acceleration Subsystem (MASS) libraries of tuned mathematical intrinsic functions. MASS libraries are thread-safe and offer improved performance over the corresponding `libm` routines. Moreover, the MASS libraries can be used without requiring code changes. The compiler supports 32-bit and 64-bit versions of the MASS vector library, `libmassvp4.a` and `libmassvp4_64.a`, respectively, which contain vector routines for single-precision and double-precision reciprocal and square root functions.

Utilities and commands

XL C/C++ Advanced Edition provides the following specialized commands to aid program development. For more information, refer to *XL C/C++ Compiler Reference*.

vac_configure Utility
> A program that creates the configuration file vac.cfg, which specifies the location of the GNU compiler and other configuration information. The C and C++ compilers use the vac.cfg for its configuration.

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++.

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`).

National language support

XL C/C++ provides support for the Unicode standard, multibyte characters, UTF-16 and UTF-32 string literals, multiple loaded locales, and bidirectionality. These features make possible or facilitate the creation of international applications.

Related References
- ″The Unicode Standard″ in *XL C/C++ Language Reference*
- ″National Languages Support″ in *XL C/C++ Compiler Reference*

Documentation and online help

XL C/C++ Advanced Edition provides product documentation in the following formats:
- Readme files.
- Installable man pages.
- An HTML-based help system.
- PDF documents.

These items are located or accessed as follows:

Readme files	The readme files are located in /opt/ibmcmp/vacpp/7.0 and in the root directory of the installation CD.
Man pages	Man pages are provided for the compiler invocations and all command-line utilities provided with the product.
HTML-based help system	A help system composed of HTML files is provided. The help system is also available online as part of the product Information Center.
PDF documents	The PDF files are located in the /opt/ibmcmp/pdf directory. They are viewable and printable from the Adobe Acrobat Reader. If you do not have the Adobe Acrobat Reader installed, you can download it from http://www.adobe.com.

The complete library of XL C/C++ PDF documents consists of the following files:

install.pdf

    XL C/C++ Installation Guide contains instructions for installing the compiler and enabling the man pages.

getstart.pdf

    Getting Started with XL C/C++ contains an overview of XL C/C++ components, explanation of new features, how-to information on customizing the compilation environment and process, summary tables of the compiler options arranged by category, an introduction to performance optimization and tuning, and general advice for porting an application to the Linux platform.

language.pdf

    XL C/C++ Language Reference contains information about the IBM implementations of the C and C++ programming languages, including the implementation-defined extensions for porting an application originally developed with GNU C and C++.

compiler.pdf

    XL C/C++ Compiler Reference contains information about the various compiler options, pragmas, macros, and built-in functions, including those used for parallel processing.

proguide.pdf

    XL C/C++ Programming Guide contains information about programming using XL C/C++ not covered in other publications.

license.pdf

    IBM XL C/C++ Advanced Edition V7.0 for Linux License Information contains information about the product license.

Accessing additional information

For the latest information about XL C/C++, visit the product documentation and support pages at the following URLs. In addition, IBM Redbooks, developed by the IBM Technical Support Organization, contain technical information based on realistic scenarios from practical experience.

- The Information Center at http://www.ibm.com/software/awdtools/vacpp/library.
- The product support site at http://www.ibm.com/software/awdtools/ccompilers.
- IBM Redbooks at http://www.redbooks.ibm.com.

You might find the following Redbooks useful for application development with XL C/C++:

__   *POWER4 Processor Introduction and Tuning Guide*, SG24-7041-00.

__   *Understanding IBM eServer pSeries Performance and Sizing*, SG24-4810-01.

What's new in version 7

XL C/C++ Advanced Edition now supports the following operating systems:
- SUSE LINUX Enterprise Server 9.0 (SLES 9)
- Red Hat Enterprise Linux 3.0 Update 3 (RHEL 3 U3)
- Red Hat Enterprise Linux 4.0 (RHEL 4)
- Y-HPC (Terra Soft's 64-bit High Performance Computing OS for PowerPC)

The new features and enhancements in XL C/C++ Advanced Edition fall into three categories: performance and optimization, conformance to industry standards, and ease of use.

Performance and optimization

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

Machine architecture and hardware

Refinements to options -qarch and -qtune

The compiler option `-qarch` controls the particular instructions that are generated for the specified machine architecture. Option `-qtune` 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. Skillful use of these options in combination is key to achieving maximal exploitation of IBM processors and hardware. The coordination of these options has been enhanced in this release to add support for the POWER5 and PowerPC 970 hardware platforms and for greater ease of use.

For a particular architecture specified by `-qarch`, compiling with the default `-qtune` suboption generates code that gives the best performance for that architecture. Option `-qarch` can now specify a group of architectures; compiling with `-qtune=auto` generates code that runs on all of the architectures in the specified group, but the instruction sequences will be those with the best performance on the architecture of the compiling machine.

The default setting for `-qarch` depends on the platform that you are using:
- If you are using SLES 9, RHEL 3 U3, or RHEL 4, the default setting is now `-qarch=ppc64grsq`.
- If you are using Y-HPC, the default setting is now `-qarch=ppc970`.

AltiVec (VMX) support

XL C/C++ for Linux supports the AltiVec programming model and provides additional features to ensure maximum compatibility with GNU C and C++ compilers. The AltiVec data types and related operations are available in 32- and 64-bit modes, wherever the architecture supports the PowerPC SIMD extension (also known as the VMX engine). The SIMD (Single Instruction, Multiple Data) instruction set enables higher utilization of microprocessor hardware. The compiler provides the ability to automatically enable SIMD vectorization at higher levels of optimization.

Built-in functions supported for POWER5 processors

Support on POWER5 processors has been added for the following built-in functions. All supported built-in functions are described in *XL C/C++ Compiler Reference*.

Built-in functions for POWER5 processors

Function	Description
`int __popcnt4(unsigned int);`	Returns the number of bits set (=1) for a 32–bit integer.
`int __popcnt8 (unsigned long long);`	Returns the number of bits set (=1) for a 64–bit integer.
`unsigned long __popcntb (unsigned long);`	Counts the 1 bits in each byte of the source operand and places that count into the corresponding byte of the result.
`int __poppar4(unsigned int);`	Returns 1 if an odd number of bits is set for a 32-bit integer. Otherwise, returns 0.
`int __poppar8 (unsigned long long);`	Returns 1 if an odd number of bits is set for a 64-bit integer. Otherwise, returns 0.
`double __fre(double);`	Returns the result of a floating-point reciprocal operation. The result is a double precision estimate of `1/x`.
`float __frsqrtes(float);`	Returns the result of a reciprocal square root operation. The result is a single precision estimate of the reciprocal of the square root of `x`.
`unsigned long __mfspr(const int);`	Return a value in the specified special purpose register.
`void __mtspr(const int, unsigned long);`	Set the special purpose register specified by `const int`.
`unsigned long __mfmsr();`	Return the machine state register.
`void __mtmsr(unsigned long);`	Set the machine state register.
`void __protected_unlimited_stream_set_go(unsigned int direction, const void* addr, unsigned int ID);`	Establish a protected stream of unlimited length that uses the identifier `ID`. The stream identifier should be within the range of 0 to 15. The stream begins with the cache line at `addr`. The stream fetches from either incremental memory addresses or decremental memory addresses, as specified by `direction`. For incremental memory addresses (that is, a forward direction), the value of `direction` is 1; for decremental memory addresses, the value of `direction` is 3. The stream is protected from being replaced by any hardware-detected streams.
`void __protected_stream_set(unsigned int direction, const void* addr, unsigned int ID);`	Establish a protected stream of limited length that uses the identifier `ID`. The stream begins with the cache line at `addr` and subsequently fetches from either incremental memory addresses or decremental memory addresses, as specified by `direction`. The stream is protected from being replaced by any hardware-detected streams.
`void __protected_stream_count(unsigned int unit_cnt, unsigned int ID);`	Set the number of cache lines for the limited-length protected stream identified by `ID`. The number of cache lines is specified by the parameter `unit_cnt` and should be within the range of 0 to 1023.
`void __protected_stream_go();`	Start to prefetch all limited-length protected streams.
`void __protected_stream_stop(unsigned int ID);`	Stop prefetching the protected steam identified by `ID`.
`void __protected_stream_stop_all();`	Stop prefetching all protected steams.

New built-in functions for floating-point division

Four new built-in functions for floating-point division are included in this release. These software implementations of floating-point division algorithms take advantage

of the PowerPC architecture and can be significantly faster than corresponding hardware instructions when used in a vector context. The new built-ins are supported for all PowerPC processors, including POWER5.

Hardware division instructions are obtained by default if floating-point division is coded in the source program, but the compiler makes the choice between the hardware or software division code, depending on which it deems faster. The new built-in functions allow the user to explicitly invoke the software algorithms. The default rounding mode (round-to-nearest) must be in effect when the routines are called.

Built-in functions for floating-point division

Function	Description
`double __swdiv_nochk(double, double);`	Floating-point division of **double** types; no range checking. Argument restrictions: numerators equal to infinity are not allowed; denominators equal to infinity, zero, or denormalized are not allowed; the quotient of numerator and denominator may not be equal to positive or negative infinity.
`double __swdiv(double, double);`	Floating-point division of **double** types. No argument restrictions.
`float __swdivs_nochk(float, float);`	Floating-point division of **float** types; no range checking. Argument restrictions: numerators equal to infinity are not allowed; denominators equal to infinity, zero, or denormalized are not allowed; the quotient of numerator and denominator may not be equal to positive or negative infinity.
`float __swdivs(float, float);`	Floating-point division of **double** types. No argument restrictions.

New XL C/C++ pragmas

Pragma directives are described in detail in *XL C/C++ Compiler Reference*.

Pragma	Description
`#pragma novector`	Prohibits the compiler from automatically vectorizing the loop that immediately follows it. Automatic vectorization refers to converting certain operations that are performed in a loop on successive elements of an array, into a call to a routine that computes several results at a time.
`#pragma nosimd`	Prohibits the compiler from automatically generating VMX instructions in the loop that immediately follows it.
`#pragma unrollandfuse`	A pragma for optimizing nested **for** loops. Instructs the compiler to replicate the body of the outer loop, which is itself a loop nest, and fuses the replicas into a single unrolled loop nest.
`#pragma stream_unroll`	Breaks a stream contained in a **for** loop into multiple streams. Intended for loops that have a large iteration count and a small number of streams.
`#pragma block_loop`	Instructs the compiler to create a blocking loop for a specific **for** loop in a loop nest. Blocking a loop involves dividing the iteration space of a loop into parts or blocks. An additional outer loop is created, known as the *blocking loop*, which drives the original loop for each block.

Pragma	Description
`#pragma loopid`	Marks a **for** loop with a scope-unique identifier. The identifier can be used by `#pragma block_loop` and others to control the transformations on that loop and to provide information on the loop transformations through the use of option `-qreport`.
`#pragma disjoint`	C++ implementation added.
extensions to `#pragma unroll`	Loop unrolling consists of replicating the body of a loop in order to reduce the number of iterations required to complete the loop. The #pragma unroll directive indicates to the compiler that the **for** loop that immediately follows the directive can be unrolled. The functionality of this pragma has been extended to allow it to be applied to both the innermost and outermost **for** loops. The extended #pragma functionality still excludes application to **for** loops that have alternate entry points.

New optimization utilities

This release contains two new utilities related to the profile-directed feedback (PDF) compilation process. Through the use of profile-directed feedback, the compiler can provide an optimized executable that reflects how that executable ran in a number of different scenarios. A PDF record is produced as a side effect of running the instrumented executable in one of these scenarios. These records constitute the data that are collated to define typical program behavior.

The **showpdf** command provides the ability to display the call and block counts for all procedures executed in a profile-directed feedback training run. The utility requires compilation under the options `-qpdf1` and `-qshowpdf`.

The **mergepdf** command allows the user to specify the relative importance of two or more PDF records and to combine them into a single record. This allows the user to compensate for training runs with higher execution counts (that is, longer run time), which would otherwise dominate the profile data.

Support for MASS vector libraries

The compiler adds support for the 32-bit and 64-bit mode versions of the IBM Mathematics Acceleration Subsystem (MASS) vector library: `libmassvp4.a` and `libmassvp4_64.a`, respectively. These libraries contain vector routines for single-precision and double-precision reciprocal and square root functions. The vector libraries are thread-safe and offer improved performance over the corresponding `libm` routines.

Starting with Version 7.0, the MASS libraries ship with the compiler.

Conformance to industry standards

OpenMP API V2.0 support for C, C++, and Fortran

The OpenMP Application Program Interface (API) is a portable, scalable programming model that provides a standard interface for developing multiplatform, shared-memory parallel applications in C, C++, and Fortran. The specification is defined by the OpenMP organization, a group of major computer hardware and software vendors, which includes IBM. XL C/C++ Advanced Edition is compliant

with OpenMP Specification 2.0: the compiler recognizes and preserves the semantics of the following OpenMP V2.0 elements:

- Comma delimiter for multiple clauses in the `#pragma omp` directive.
- The `num_threads` clause.
- The `copyprivate` clause.
- `threadprivate` static block scope variables.
- Support for C99 variable length arrays.
- Redundant declaration of private variables.
- Timing routines `omp_get_wtime` and `omp_get_wtick`.

Enhanced Unicode and NLS support

As recommended in a recent report from the C Standard committee, the C compiler extends C99 to add new data types to support UTF-16 and UTF-32 literals. The C++ compiler also supports these new data types for compatibility with C.

Support for Boost libraries

The XL C++ compiler delivers a high level of compatability with the 1.30.2 Boost libraries. These libraries were created to provide a set of reusable, Open Source C++ libraries that are suitable for standardization. For more information, see the Boost web site at http://www.boost.org.

Language extensions related to GNU C and C++

The GNU C extensions to C99 and the GNU C++ extensions to Standard C++ are not industry standards. Nevertheless, these non-proprietary language features from the Open Source community have attained a certain currency. XL C/C++ implements a subset of the GNU C and C++ extensions. Support for the following GNU C features has been added in this release.

Feature	Remarks
Labels as Values	Including computed **goto** statements. This feature is now fully compatible with the GNU C implementation.
Type Attributes	Attributes aligned, packed, transparent_union.
Function Attributes	Attributes format, format_arg, always_inline, noinline.
Variable Attributes	C++ support added for attribute `section`.
Alternate Keywords	Internal changes to implementation of __**extension**__.
Nested Functions	▶ C C support only.
Cast to a Union Type	▶ C C support only.
Macros with a Variable Number of Arguments	Using an identifier in place of __VA_ARGS__ and removing trailing comma when no __VA_ARGS__ arguments are specified.
gcc Inline Assembler Instructions with C Expression Operands	Partial support only.
GNU C Complex Types	C++ support added.
GNU C Hexadecimal Float Constants	C++ support added.
C99 Compound Literals	C++ support added.

Feature	Remarks
Arrays of Length Zero	C++ support added.
Variable Length Arrays	C++ support added.

Ease of use

New C++ compiler invocation

The compiler invocation **xlc++** has been added for portability among all supported platforms. The invocation is equivalent to the invocation **xlC** on all platforms and is recommended. However, **xlC** is still fully supported.

Documentation

A man page is provided for the compiler invocation commands and for each command-line utility. The man page for the compiler invocations replaces the text help file, which was provided in previous releases.

Template registry enhancement

The C++ compiler uses a batch template instantiation scheme that involves a registry of template instantiations. In this release, the compiler adds versioning information to the template registry file that is created. This information is used by the compiler internally to track which version of the template registry file format should be used.

New XL C/C++ options

New and changed compiler options are described in detail in the *XL C/C++ Compiler Reference*.

Option	Description and remarks
-qabi_version=*n*	Instructs the compiler to use version *n* of the C++ ABI, where n can be: • 1, which allows users to get the same C++ ABI behavior introduced in G++3.2 and found in G++3.3. • 2, which allows users to get the same C++ ABI behavior introduced in G++3.4. The default depends on the operating system: • **1** on SLES 9, RHEL 3 U3, and Y-HPC • **2** on RHEL 4
-qaltivec	Enables compiler support for AltiVec data types. The default is **-qnoaltivec**.
-qasm=gcc	Enables partial support for assembler instructions with C expression operands. Instructs the compiler to recognize the **asm** keyword and its alternate spellings and to use the gcc syntax and semantics for the keyword. The default is **-qnoasm**.
-qasm_as	Specifies the path and flags used to invoke an alternate assembler program in order to handle the code in an **asm** directive. This option overrides the default setting of the **as** command defined in the compiler configuration file.

Option	Description and remarks
-qdirectstorage	Asserts that write-through enabled or cache-inhibited storage may be referenced in a given compilation unit. The intention of this option is to avoid unexpected behavior due to different storage control attributes that are allowed by the PowerPC architecture. The default is **-qnodirectstorage**.
-qenablevmx	Instructs the compiler to generate VMX (AltiVec) code in any compiler phase. This option ensures the correct default setting of **-qaltivec** for the operating system in the development environment. For RHEL 3 U3 and Y–HPC, the default is **-qnoenablevmx**. For RHEL 4 and SLES 9, the default is **-qenablevmx**.
-qkeepparm	Ensures that the parameters of a function passed in registers are saved onto the stack, instead of possibly being moved to different memory locations to improve performance. The default is **-qnokeepparm**.
-qipa=threads[=N] | nothreads	Instructs the optimizer to create N threads and run up to N backends in parallel, where N is an integer in the range 1–MAXINT. The default is **nothreads** suboption, which is equivalent to running a single serial process. You can use this feature to reduce the IPA link time on multiprocessor computers.
-qnoprefetch	Instructs the compiler not to automatically insert software prefetch instructions, thus allowing the user to turn off this aspect of optimization. The default is **-qprefetch**.
-qnotrigraph	Instructs the compiler not to interpret trigraph sequences, regardless of the specified language level. On Linux, the default is **-qtrigraph**.
-qsaveopt	Instructs the compiler to save the command-line options against which a source file is compiled into the corresponding object file. The option has no effect if compilation does not result in a .o file. The default is **-qnosaveopt**.
-qshowpdf	When specified with **-qpdf1** and a an optimization level of **-O** or higher, the compiler inserts additional profiling information into the compiled application to collect call and block counts for all procedures in the application. Running the compiled application records the call and block counts to the file ._pdf. The contents of ._pdf can then be retrieved with the **showpdf** utility. The default is **-qnoshowpdf**.
-qsourcetype	Controls the interpretation of input file names. The default behavior is that the programming language of a source file is implied by the suffix of its file name. The default is **-qsourcetype**.
-qutf	Enables the recognition of UTF literal syntax which provides 16-bit and 32-bit string literals for Unicode encoding forms.
-qflttrap=nanq	Instructs the compiler to generate extra instructions in the code to trap NaNQs (Not a Number Quiet). The intent is to detect all NaNQs handled by or generated by floating point instructions, including those created by valid operations.
-qhot=simd	Instructs the compiler to attempt automatic SIMD vectorization. The default is **-qhot=nosimd**.
-qipa=infrequentlabel	Specifies a list of labels that are likely to be called infrequently during the course of a typical program run. The compiler can make other parts of the program faster by doing less optimization for calls to these labels. This option is only applicable to user-defined labels.

Customizing the compilation environment

This section discusses the mechanisms used by XL C/C++ to specify the search paths for directories containing include files, libraries, and the location of a GNU C or C++ compiler. These mechanisms are environment variables, include files, attributes in a configuration file, and command-line options. The vac_configure utility is provided to facilitate the creation of valid configuration files.

The important search paths for XL C/C++ are the standard directory locations for:
- A 32-bit GNU compiler, a 64-bit GNU compiler, or both
- GNU C include files
- GNU C++ include files
- IBM C and C++ headers
- GNU C library paths

Related Reference
- For more information on vac_configure, see *XL C/C++ Installation Guide*.

Environment variables

Part of the compilation environment are the search paths for special files such as libraries and include files. The following system variables are used by the compiler.

LD_LIBRARY_PATH
> Specifies the directory path for dynamically loaded libraries. Used by the GNU linker at link time and at run time.

LD_RUN_PATH
> Specifies additional directory paths to be searched at run time for dynamically loaded libraries. The setting does not affect the search paths used by the GNU linker at link time.

MANPATH   Specifies the directory path for the product man pages.

NLSPATH   Specifies the directory path of National Language Support libraries.

PATH   Specifies the directory path for the executable files of the compiler.

PDFDIR   Specifies the directory in which the profile data file is created. The default value is unset, and the compiler places the profile data file in the current working directory. Setting this variable to an absolute path is recommended for profile-directed feedback.

TMPDIR   Specifies the directory in which temporary files are created. The default location may be inadequate at high levels of optimization, where temporary files can require significant amounts of disk space.

Setting up the environment for the invocation commands

The command-line interfaces for XL C/C++ are not automatically installed in /usr/bin. To invoke the compiler without having to specify the full path, do one of the following steps:
- Create symbolic links for the specific driver contained in /opt/ibmcmp/vacpp/7.0/bin and /opt/ibmcmp/vac/7.0/bin to /usr/bin.
- Add /opt/ibmcmp/vacpp/7.0/bin to the PATH environment variable.

Ensure the correct NLSPATH for the message catalogs

The NLSPATH environment variable informs the compiler how to find the appropriate message catalogs.

To ensure that the path is correct, issue the following command:

```
export NLSPATH=$NLSPATH:smprt-path/msg/%L/%N:
  compiler-path/vacpp/6.0/msg/%L/%N
```

where *smprt-path* and *compiler-path* are the installation locations specified when you installed the packages.

Note: If the default installation location is used, then *smprt-path* and *compiler-path* will both be `/opt/ibmcmp`.

Include files

The locations for GNU, IBM, and system header files are most conveniently specified in a configuration file.

The compiler option **-I** *directory_name* allows you to add directories to the search paths in the configuration file. The configuration file itself uses the **-I** option internally to set the directory paths that it controls. The compiler searches the directories specified by **-I** within the configuration file before searching those specified by **-I** options on the command line.

See *XL C/C++ Compiler Reference* for more information.

Configuration files

A configuration file is a plain text file that specifies options that are read every time you run the compiler. The name of a configuration file ends with a .cfg file name extension.

After you have a default configuration file (/etc/opt/ibmcmp/vac/7.0/vac.cfg), you can create others. XL C/C++ provides a template, which was used to create /etc/opt/ibmcmp/vac/7.0/vac.cfg. However, any existing configuration file can be used as the template for creating another with the vac_configure utility.

You can instruct the compiler to use a particular configuration file by invoking the compiler with the **-F** option and specifying the fully qualified file name.

Command-line options

The compiler options controlling the standard include paths are shown in the table below. The value for *paths* is specified by the user on the command line. When a configuration file is used, the value for *paths* is the value of the attribute named in the second column. The configuration file used by default is that which was created by the vac_configure utility. The compiler processes each attribute in the configuration file to create the corresponding option that ensures the proper search path for include files.

Linux-specific configuration options and related attribute, 32- and 64-bit modes

Option name	Attribute	Usage	Conflict resolution
-qgcc_c_stdinc=<*paths*>	gcc_c_stdinc gcc_c_stdinc_64	Specifies the search locations for the gcc headers. No default value.	When specified multiple times in the same command, the last one prevails. The option is ignored if the **-qnostdinc** option is in effect.
-qgcc_cpp_stdinc=<*paths*>	gcc_cpp_stdinc gcc_cpp_stdinc_64	Specifies the search locations for the g++ headers. No default value.	When specified multiple times in the same command, the last one prevails. The option is ignored if the **-qnostdinc** option is in effect.
-qc_stdinc=<*paths*>	xlc_c_stdinc xlc_c_stdinc_64	Specifies the search locations for standard include files for the IBM C headers. No default value.	When specified multiple times in the same command, the last one prevails. The option is ignored if the **-qnostdinc** option is in effect.
-qcpp_stdinc=<*paths*>	xlc_cpp_stdinc xlc_cpp_stdinc_64	Specifies the search locations for the IBM C++ headers. No default value.	When specified multiple times in the same command, the last one prevails. The option is ignored if the **-qnostdinc** option is in effect.

Related References
- ″Compiler command line options″ in *XL C/C++ Compiler Reference*

Controlling the compilation process

The overall compilation process consists of three phases: preprocessing, translation to object code, and linking. By default, a compiler invocation command invokes all phases of the compilation process to translate a program from source code to executable output. If file names for input and output files are specified when the compiler is invoked, it determines the starting and ending phases from the file name suffix (extension) of the input and output files.

You can also create a particular type of output file at any compilation phase by using appropriate compiler options. For example, invoking the **xlc** or **xlC** command with the **-E** or **-P** option performs only the preprocessing phase on the input files. The compiler invocation determines from the extension of the input file name whether to call the IBM compiler, the assembler, or the linker. The assembler and linker are tools supplied with the Linux operating system.

Related References
- -qphsinfo compiler option in *XL C/C++ Compiler Reference*

Invoking the compiler

XL C/C++ provides a selection of base compiler invocation commands, which support various version levels of the C and C++ languages. Each invocation command automatically sets a compiler suboption for language level, options for other related language features, and any related predefined macros. In most cases, you should use the **xlc** command to compile C source files and the **xlC** command to compile C++ source files, or when you have both C and C++ source files. The variations of the base command are provided to support the requirements of threaded applications and are formed by attaching the suffix _r to a base command.

Supported invocation commands

xlc	xlc_r	The _r invocations are used
xlc++	xlc++_r	for compiling thread-safe
xlC	xlC_r	applications, also referred to
cc	cc_r	as *reentrant compiler*
c89	c89_r	*invocations*.
c99	c99_r	

The invocation **xlc++** is equivalent to **xlC**.

In addition, the gxlc and gxlc++ utilities are specialized compiler invocations.

Object model

Only one object model exists for the Linux platform. If you are porting an application that relies on an object model supported on AIX (`ibm` or `classic`), you might need to modify your code. The compiler options and pragmas for specifying a particular object model are not available for Linux.

Types of input and output files

The compiler uses the file name extension to determine the appropriate compilation phase and invoke the associated tool.

The compiler accepts the following types of files as input:

Accepted input file types

File type description	File name extension	Example
C and C++ source file	.c (lowercase *c*) for C language source files; .C (uppercase *c*), .cc, .cp, .cpp, .cxx, .c++ for C++ source files	*file_name*.c *file_name*.C, *file_name*.cc, *file_name*.cpp, *file_name*.cxx, *file_name*.c++
Preprocessed source file	.i	*file_name*.i
Object file	.o	`hello.o`
Assembler file	.s	`check.s`
Archive file	.a	`v1r5.a`
Loadable module or shared library file	.so	`my_shrlib.so`
IPA control files (**-qipa=***file_name*)	No naming convention for *file_name* is enforced.	`ipa.ctl`

You can specify the following types of output files when invoking the compiler:

Types of output files

File type description	Example
Executable file	By default, `a.out`
Object files	*file_name*.o
Loadable module or shared object file	*file_name*.so
Assembler files	*file_name*.s
Preprocessed files	*file_name*.i
Listing files	*file_name*.lst
Target file	*file_name*.d

Related References
- `-qsourcetype` compiler option in *XL C/C++ Compiler Reference*

Default behavior

If you invoke the compiler without specifying any options, the behavior of the compiler is governed by the following default settings:
- Attempts to read and invoke the options specified in a configuration file.
- Searches for library files in directories in the LD_LIBRARY_PATH variable.
- Aligns structures using the default alignment, `-qalign=linuxppc`.
- Produces an unoptimized executable named `a.out` in the current directory.
- Diagnoses AltiVec programming constructs as syntax errors.

See *XL C/C++ Compiler Reference* for more information.

Getting started with 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, in a configuration file, in your source code, or any combination of these techniques. Most options that are not explicitly set take the default settings.

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 applies the following priority sequence unless otherwise specified:

1. Source file *overrides*
2. Command line *overrides*
3. Configuration file *overrides*
4. Default settings

Generally, among multiple command-line options, the last 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).

Related References
- See *XL C/C++ Compiler Reference* for more information.

Compiler messages

XL C/C++ uses a five-level classification scheme for diagnostic messages. Each level of severity is associated with a compiler response. Not every error halts compilation. The following table provides a key to the abbreviations for the severity levels and the associated compiler response.

Severity levels and compiler response

Letter	Severity	Compiler Response
I	Informational	Compilation continues. The message reports conditions found during compilation.
W	Warning	Compilation continues. The message reports valid, possibly unintended conditions.
C E	Error	Compilation continues and object code is generated. Error conditions exist that the compiler can correct, but the program might not produce the expected results.
S	Severe error	Compilation continues, but object code is not generated. Error conditions exist that the compiler cannot correct.

Severity levels and compiler response

Letter	Severity	Compiler Response
U	Unrecoverable error	The compiler halts. An unrecoverable error has been encountered. If the message indicates a resource limit (for example, file system full or paging space full), provide additional resources and recompile. If it indicates that different compiler options are needed, recompile using them. If the message indicates an internal compiler error, the message should be reported to your IBM service representative.

The default behavior of the compiler is to compile with the option **-qnoinfo** or **-qinfo=noall**. The suboptions for **-qinfo** provide the ability to specify a particular category of informational diagnostics. For example, **-qinfo=por** limits the output to those messages related to portability issues.

Note: In C, the option **-qinfo** specified without suboption is equivalent to **-qinfo=all**; in C++, **-qinfo** specified without suboption is equivalent to **-qinfo=all:noppt**.

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 one of the return codes documented in *XL C/C++ Compiler Reference*.

Compiler message format

By default, diagnostic messages have the following format:

```
"file", line line_number.column_number: 15cc-nnn (severity) message_text.
```

where 15 is the compiler product identifier, *cc* is a two-digit code indicating the compiler component that issued the message, *nnn* is the message number, and *severity* is the letter of the severity level. The possible values for *cc* are:
00 Code generating or optimizing message
01 Compiler services message
05 Message specific to the C compiler
06 Message specific to the C compiler
40 Message specific to the C++ compiler
86 Message specific to interprocedural analysis (IPA)

This format is the same as compiling with the **-qnosrcmsg** option enabled. To get an alternate message format in which the source line displays with the diagnostic message, try compiling with **-qsrcmsg** option. Enabling this option instructs the compiler to print to standard error the source line where the compiler thinks the error lies; a second line below it, whenever possible, that points to a specific point in that source line; and the diagnostic message.

Note: Messages are not intended to be used as input to other programs. The message format and content are not intended to be a programming interface and may change from release to release.

Platform-specific options

This section features options that are basic to using the compiler on the Linux platform.

See *XL C/C++ Compiler Reference* for more information.

Selected compiler options specific to the Linux platform

Option name	Description
-qgcc_c_stdinc=<*paths*>	Specifies the directory search paths for the GNU C headers.
-qgcc_cpp_stdinc=<*paths*>	Specifies the directory search paths for the GNU C++ headers.
-qc_stdinc=<*paths*>	Specifies the directory search paths for the IBM C headers.
-qcpp_stdinc=<*paths*>	Specifies the directory search paths for the IBM C++ headers.

The following options provide specialized control of directory search paths. The options are cumulative, rather than preemptive. The paths specified on the command line with the options **-L**, **-R**, and **-l** (lowercase *L*) will have lower priority at link time than those specified as an option in the configuration file, but higher priority than paths specified as an attribute in the configuration file.

Selected path control options

Option name	Description
-I	Specifies additional directory paths to be searched for `#include` files.
-l	Specifies the shared library or archive file for static linking.
-L	Specifies the library search paths to be searched at link time.
-R	Specifies the directory paths to be searched at run time for dynamically loaded libraries.

Related References
- ″Summary of options for optimization and performance″ in *XL C/C++ Programming Guide*

Reusing GNU C and C++ compiler options with gxlc and gxlc++

Each of the gxlc and gxlc++ utilities accepts GNU C or C++ compiler options and translates them into comparable XL C/C++ options. Both utilities use the XL C/C++ options to create an **xlc** or **xlC** invocation command, which they then use to invoke XL C/C++. These utilities are provided to facilitate the reuse of make files created for applications previously developed with GNU C and C++. However, to fully exploit the capabilities of XL C/C++, it is recommended that you use the XL C/C++ invocation commands and their associated options.

The actions of gxlc and gxlc++ are controlled by the configuration file `gxlc.cfg`. The GNU C and C++ options that have an XL C or XL C++ counterpart are shown in this file. Not every GNU option has a corresponding XL C/C++ option. gxlc and gxlc++ return warnings for input options that were not translated.

The gxlc and gxlc++ option mappings are modifiable. For information on adding to or editing the gxlc and gxlc++ configuration file, see "Configuring the option mapping" on page 26.

Example

To use the gcc `-ansi` option to compile the C version of the Hello World program, you can use:

```
gxlc -ansi hello.c
```

which translates into:

```
xlc -F:c89 hello.c
```

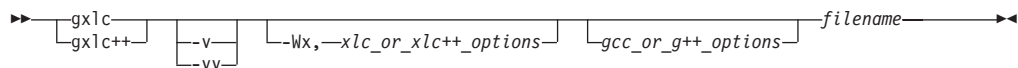This command is then used to invoke the XL C compiler.

gxlc and gxlc++ return codes

Like other invocation commands, gxlc and gxlc++ return output, such as listings, diagnostic messages related to the compilation, warnings related to unsuccessful translation of GNU options, and return codes. If gxlc or gxlc++ cannot successfully call the compiler, it sets the return code to one of the following values:

40   A gcc or g++ option error or unrecoverable error has been detected.
255   An error has been detected while the process was running.

gxlc and gxlc++ syntax

The following diagram shows the gxlc and gxlc++ syntax:

```
►►──┬─gxlc──┬──┬────┬──┬───────────────────────────┬──┬─────────────────┬──filename──────────►◄
    └─gxlc++─┘  ├─-v─┤  └─-Wx,──xlc_or_xlc++_options─┘  └─gcc_or_g++_options─┘
               └─-vv─┘
```

where:

filename
        Is the name of the file to be compiled.

-v   Allows you to verify the command that will be used to invoke XL C/C++. gxlc or gxlc++ displays the XL C/C++ invocation command that it has created, before using it to invoke the compiler.

-vv   Allows you to run a simulation. gxlc or gxlc++ displays the XL C/C++ invocation command that it has created, but does not invoke the compiler.

-Wx,`xlc_or_xlc++_options`
        Sends the given XL C/C++ options directly to the `xlc` or `xlc++` invocation command. gxlc or gxlc++ adds the given options to the XL C/C++ invocation it is creating, without attempting to translate them. Use this option with known XL C/C++ options to improve the performance of the utility. Multiple *xlc_or_xlc++_options* use a comma delimiter.

gcc_or_g++_options
        Are the gcc or g++ options that are to be translated to `xlc` or `xlc++` options. The utility emits a warning for any option it cannot translate. The gcc and g++ options that are currently recognized by gxlc and gxlc++ are listed in the configuration file `gxlc.cfg`. Multiple *gcc_or_g++_options* are delimited by the space character.

GNU C and C++ to XL C/C++ option mapping

The following table lists the GNU C and C++ options that are accepted and translated by gxlc and gxlc++. All other GNU options that are specified as input to one of these utilities are ignored or generate an error. If the negative form of a GNU option exists, then the negative form is also recognized and translated by gxlc and gxlc++.

Mapped options: GNU C and C++ to XL C/C++

| GNU C and C++ option	XL C/C++ option
-###	-#
-ansi	-F:c89
-B	-B
-C	-C
-c	-c
-D*macro[=defn]*	-D*macro[=defn]*
-E	-E
-e	-e
-fdollars-in-identifiers	-qdollar
C++ -fdump-class-hierarchy	-qdump_class_hierarchy
C++ -fexceptions	-qeh
C++ -ffor-scope	-qlanglvl=ansifor
C++ -fno-for-scope	-qlanglvl=noansifor
-ffunction-sections	-qfuncsect
-finline	-qinline
-finline-functions	-qinline **Note:** The **-qinline=limit=n** is not a valid suboption in XL C/C++. The **-finline-limit=n** GNU C and C++ suboption cannot be mapped to this XL C/C++suboption.
C++ -fkeep-inline-functions	-qkeepinlines
C++ -fno-gnu-keywords	-qnokeyword=typeof
C++ -fno-operator-names	-qnokeyword=and -qnokeyword=bitand -qnokeyword=bitor -qnokeyword=compl -qnokeyword=not -qnokeyword=or -qnokeyword=xor
-fpascal-strings	-qmacpstr
-fPIC	-qpic=large
-fpic	-qpic=small
C++ -frtti	-qrtti
-fshort-enums	-qenum=small
-fsigned-bitfields	-qbitfields=signed
-fsigned-char	-qchars=signed
-fstrict-aliasing	-qalias=ansi
-fsyntax-only	-qsyntaxonly

Mapped options: GNU C and C++ to XL C/C++

GNU C and C++ option	XL C/C++ option
-funroll-all-loops	-qunroll=yes
-funroll-loops	-qunroll=yes
-funsigned-bitfields	-qbitfields=unsigned
-funsigned-char	-qchars=unsigned
-fwritable-strings	-qnoro
-g	-g
-g3	-g
-gdwarf	-g
-ggdb	-g
-I*dir*	-I*dir*
-L*dir*	-L*dir*
-l*library*	-l*library*
-M	-M
-MD	-M
-mcpu=power	-qarch=pwr
-mcpu=powerpc	-qarch=ppc
-mcpu=powerpc64	-qarch=ppc64
-mno-fused-madd	-qfloat=nomaf
-mfused-madd	-qfloat=maf
-mlong-double-64	-qnolonglong
-mlong-double-128	-qlonglong
-mpower	-qarch=pwr
-mpowerpc	-qarch=ppc
-mpowerpc64	-qarch=ppc64
-mtune=power	-qtune=pwr
-mtune=powerpc	-qtune=ppc
-mtune=powerpc64	-qtune=ppc64
-nodefaultlibs	-qnolib
-nostartfiles	-qnocrt
-nostdinc	-qnostdinc
-nostdlib	-qnolib -qnocrt
-O	-O
-O0	-qnoopt
-O1	-O
-O2	-O2
-O3	-O3
-Os	-O2 -qcompact
-o	-o
-p	-p
-pg	-pg

Mapped options: GNU C and C++ to XL C/C++

GNU C and C++ option	XL C/C++ option
-r	-r
-S	-S
-s	-s
C -std=c89	-F:c89
C -std=iso9899:1990	-F:c89
C -std=iso9899:199409	-F:c89
C -std=c99	-F:c99
C -std=c9x	-F:c99
C -std=iso9899:1999	-F:c99
C -std=iso9899:199x	-F:c99
C -std=gnu89	-qlanglvl=extc89
C -std=gnu99	-qlanglvl=extc99
C -std=gnu9x	-qlanglvl=extc99
C++ -std=c++98	-qlanglvl=strict98
C++ -std=gnu++98	-qlanglvl=extended
-time	-qphsinfo
-trigraphs	-qtrigraph
-U*macro*	-U*macro*
-u	-u
-Wformat	-qformat
-Wuninitialized	-qinfo=ini
-Wunreachable-code	-qinfo=eff
-Wa,*option*	-Wa,*option*
-Wl,*option*	-Wl,*option*
-Wp,*option*	-Wp,*option*
-w	-w
-x assembler	-qsourcetype=assembler
-x c	-qsourcetype=c
-x c++	-qsourcetype=c++
-x none	-qsourcetype=default
-Z	-Z

All other GNU options are ignored and issue an informational message.

Configuring the option mapping

The gxlc and gxlc++ utilities use the configuration file `gxlc.cfg` to translate GNU C and C++ options to XL C/C++ options. Each entry in `gxlc.cfg` describes how the utility should map a GNU C or C++ option to an XL C/C++ option and how to process it.

An entry consists of a string of flags for the processing instructions, a string for the GNU C option, and a string for the XL C/C++ option. The three fields must be separated by whitespace. If an entry contains only the first two fields and the XL C/C++ option string is omitted, the GNU C option in the second field will be recognized by gxlc and silently ignored.

The # character is used to insert comments in the configuration file. A comment can be placed on its own line, or at the end of an entry.

The following syntax is used for an entry in `gxlc.cfg`:

abcd   `"gcc_or_g++_option"`   `"xlc_or_xlc++_option"`

where:

a   Lets you disable the option by adding `no-` as a prefix. The value is either `y` for yes, or `n` for no. For example, if the flag is set to `y`, then `finline` can be disabled as `fno-inline`, and the entry is:

       `ynn*`   `"-finline"`   `"-qinline"`

       If given `-fno-inline`, then gxlc will translate it to `-qnoinline`.

b   Informs the utility that the XL C/C++ option has an associated value. The value is either `y` for yes, or `n` for no. For example, if option `-fmyvalue=`*n* maps to `-qmyvalue=`*n*, then the flag is set to `y`, and the entry is:

       `nyn*`   `"-fmyvalue"`   `"-qmyvalue"`

       gxlc and gxlc++ will then expect a value for these options.

c   Controls the processing of the options. The value can be:

- `n`, which tells the utility to process the option listed in the gcc-option field
- `i`, which tells the utility to ignore the option listed in the gcc-option field. gxlc and gxlc++ will generate a message that this has been done, and continue processing the given options.
- `e`, which tells the utility to halt processing if the option listed in the gcc-option field is encountered. gxlc and gxlc++ will also generate an error message.

       For example, the gcc option `-I-` is not supported and must be ignored by gxlc and gxlc++. In this case, the flag is set to `i`, and the entry is:

       `nni*`   `"-I-"`

       If gxlc and gxlc++ encounters this option as input, it will not process it and will generate a warning.

d   Lets gxlc and gxlc++ include or ignore an option based on the type of compiler. The value can be:

- `c`, which tells gxlc and gxlc++ to translate the option only for C.
- `x`, which tells gxlc and gxlc++ to translate the option only for C++.
- `*`, which tells gxlc and gxlc++ to translate the option for C and C++.

For example, `-fwritable-strings` is supported by both compilers, and maps to `-qnoro`. The entry is:

```
nnn*        "-fwritable-strings"        "-qnoro"
```

"gcc_or_g++_option"
> Is a string representing a gcc or g++ option supported by GNU C, Version 3.3. This field is required and must appear in double quotation marks.

"xlc_or_xlc++_option"
> Is a string representing an XL C/C++ option. This field is optional, and, if present, must appear in double quotation marks. If left blank, gxlc and gxlc++ ignores the *gcc_or_g++_option* in that entry.

It is possible to create an entry that will map a range of options. This is accomplished by using the asterisk (`*`) as a wildcard. For example, the gcc `-D` option requires a user-defined name and can take an optional value. It is possible to have the following series of options:

```
-DCOUNT1=100
-DCOUNT2=200
-DCOUNT3=300
-DCOUNT4=400
```

Instead of creating an entry for each version of this option, the single entry is:

```
nnn*        "-D*"                       "-D*"
```

where the asterisk will be replaced by any string following the `-D` option.

Conversely, you can use the asterisk to exclude a range of options. For example, if you want gxlc or gxlc++ to ignore all the `-std` options, then the entry would be:

```
nni*        "-std*"
```

When the asterisk is used in an option definition, option flags *a* and *b* are not applicable to these entries.

The character `%` is used with a GNU C or GNU C++ option to signify that the option has associated parameters. This is used to insure that gxlc or gxlc++ will ignore the parameters associated with an option that is ignored. For example, the `-include` option is not supported and uses a parameter. Both must be ignored by the application. In this case, the entry is:

```
nni*        "-include %"
```

Related References
- The GNU Compiler Collection online documentation at `http://gcc.gnu.org/onlinedocs/`

Options summary: C compiler

This chapter appendix presents a summary of the C compiler options, grouped by type. The higher level groupings contain subgroups of options. In addition to a subgroup for basic translation of source code, one subgroup comprises options for special handling or control of the code, such as adding specialized debugging information. Another subgroup pertains to control of the linker and library search paths. Options related to performance and optimization are summarized at the end of chapter "Getting started with optimization" on page 33. For description, full option syntax, and usage of each option, see *XL C/C++ Compiler Reference*.

Basic translation

The options in this grouping have the broadest applicability for basic translation of source code. The subgroups of compiler options are generally concerned with:

- Standards compliance.
- Compilation mode or control of the compiler driver.
- Manipulating the source code for code generation.
- Generating specialized diagnostics.
- Manipulating the compiled code.

Options related to basic translation of source code

Standards compliance	Compilation mode or control of compiler driver
-qgenproto, -qnogenproto -qlanglvl -qlibansi, -qnolibansi	-# -q32 -q64 -qabi_version=n, -qaltivec, -qnoaltivec -F -qpath -qproto, -qnoproto -qsourcetype
Control of the preprocessor	
-C -D -E -P -U	
Source code generation	
-qalloca -qasm, -qnoasm -qasm_as -qattr, -qnoattr -B -C -qcpluscmt, -qnocpluscmt -D -qdbcs, -qnodbcs -qdigraph, -qnodigraph -qdirectstorage, -qnodirectstorage -E -qenablevmx -qfuncsect, -qnofuncsect -qignprag -M	-qmakedep -qmbcs, -qnombcs -qminimaltoc, -qnominimaltoc -P -qsmallstack, -qnosmallstack -qsyntaxonly -t -qtabsize -qtrigraph, -qnotrigraph -U -qutf, -qnoutf -qvrsave, -qnovrsave -W
Diagnostics	**Compiled code**
-qflag -qinfo, -qnoinfo -qmaxerr, -qnomaxerr -qphsinfo, -qnophsinfo -qprint, -qnoprint -qshowinc, -qnoshowinc -qsource, -qnosource -qsuppress, -qnosuppress -V -v -w -qwarn64, -qnowarn64 -qxcall, -qnoxcall	-qbitfields -c -qchars -qdataimported -qdatalocal -qdollar, -qnodollar -o -qprocimported -qproclocal -qprocunknown -S -qstatsym, -qnostatsym -qtbtable -qupconv, -qnoupconv

Specialized diagnostics

The options in this grouping control the ability to generate and display information related to the compilation process.

Options enabling specialized diagnostics

Diagnostics	
-qflag -qinfo, -qnoinfo -qmaxerr, -qnomaxerr -qphsinfo, -qnophsinfo -qprint, -qnoprint -qshowinc, -qnoshowinc -qsource, -qnosource	-qsuppress, -qnosuppress -V -v -w -qwarn64, -qnowarn64 -qxcall, -qnoxcall

Special handling and control

The options in this grouping provide fine-grain control of the translation process and have less general applicability than basic translation options. The topics within this grouping of compiler options are generally concerned with:

- Data alignment.
- Compilation mode or control of the compiler driver.
- Manipulating the source code for code generation.
- Generating specialized diagnostics.
- Manipulating the compiled code.

Options for special handling, fine tuning, and debugging

Data alignment	Parallelization
-qalign -qenum	-qthreaded, -qnothreaded -qtls, -qnotls (RedHat Linux only)
Floating-point and numerical features	
Sizes -qlonglit, -qnolonglit -qlonglong, -qnolonglong	*Rounding of floating-point values* -y
Single-precision values None applicable for the Linux on pSeries platform	*Other floating-point options* -qfloat -qflttrap, -qnoflttrap
Debugging	
-qcheck, -qnocheck -qdbxextra, -qnodbxextra -qfullpath, -qnofullpath -g -qhalt -qinitauto, -qnoinitauto	-qkeepparm, -qnokeepparm -qlinedebug, -qnolinedebug -qlist, -qnolist -qlistopt, -qnolistopt -qsaveopt, -qnosaveopt -qsymtab -qxref, -qnoxref

Linking and library-related options

The options in this grouping are related to the linking phase of the compilation process. This grouping also contains options that provide specialized ways to specify search paths for finding libraries and header files. These compiler options are generally concerned with:

- Placing string literals and constants.
- Static and dynamic linking and libraries.
- Specifying search directories.

Options for controlling the ld command

Placing string literals and constants	Static and dynamic linking and libraries
-qkeyword, -qnokeyword -qro, -qnoro -qroconst, -qnoroconst	-qstdinc, -qnostdinc
Search directories	**Other linker options**
-I -L -l (lowercase el) -qc_stdinc -qcomplexgccincl, -qnocomplexgccincl -qgcc_c_stdinc -qidirfirst, -qnoidirfirst -r	-qinlglue, -qnoinglue -qcrt, -qnocrt -qlib, -qnolib

Options summary: C++ compiler

Most of the C compiler options are available for compiling C++ programs. The following table presents additional compiler options specific to compiling C++ programs and the C options that are *not* available for compiling C++ programs on the Linux platform:

Compiler options for C++ programs

C++-specific options	C-only options
-+ -qcinc -qcpp_stdinc -qeh, -qnoeh -qgcc_cpp_stdinc -qhaltonmsg -qpriority -qrtti, -qnortti -qstaticinline, -qnostaticinline -qtempinc, -qnotempinc -qtemplaterecompile, -qnotemplaterecompile -qtemplateregistry, -qnotemplateregistry -qtempmax -qtmplparse -qvftable, -qnovftable	-qc_stdinc -qcpluscmt, -qnocpluscmt -qdbxextra, -qnodbxextra -qgcc_c_stdinc -qgenproto, -qnogenproto -qproto, -qnoproto -qsyntaxonly -qupconv, -qnoupconv

Getting started with optimization

Simple compilation is a translation or transformation of the source code into an executable or shared object. An optimizing transformation is one that gives your application better overall performance at run time. XL C/C++ provides a portfolio of optimizing transformations tailored to the PowerPC architecture. These transformations can:

- Reduce the number of instructions executed for critical operations.
- Restructure the 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.

Their aim is to make your application run faster.

Significant performance improvements can be achieved with relatively little development effort if you understand the available controls that affect the transformation of well-written code. Programming models such as OpenMP allow you to write high-performance code. This section describes some of the optimizations the compiler can perform to help you balance the trade-offs among run-time performance, hand-coded micro-optimizations, general readability, and overall portability of your source code.

This discussion assumes that you have used a profiler to identify the areas in your code where optimization might be appropriate.

Optimizations are often attempted in the later phases of application development cycles, such as product release builds. If possible, you should test and debug your code without optimization before attempting to optimize it. Embarking on optimization should mean that you have chosen the most efficient algorithms for your program and that you have implemented them correctly. To a large extent, compliance with language standards is directly related to the degree to which your code can be successfully optimized. Optimizers are the ultimate conformance test!

Optimization is controlled by compiler options, directives, and pragmas. However, compiler-friendly programming idioms can be as useful to performance as any of the options or directives. It is no longer necessary nor is it recommended to excessively hand-optimize your code (for example, manually unrolling loops). Unusual constructs can confuse the compiler (and other programmers), and make your application difficult to optimize for new machines.

It should be noted that not all optimizations are beneficial for all applications. A trade-off usually has to be made between an increase in compile time, accompanied by reduced debugging capability, and the degree of optimization done by the compiler.

Related References
- ″Using optimization levels″ in *XL C/C++ Programming Guide*
- ″Optimizing your applications″ in *XL C/C++ Programming Guide*

33

Selected compiler options for optimization

The following table features a selection of basic compiler options for optimizing program performance. For an exhaustive list, see *XL C/C++ Programming Guide*. For documentation of the available suboptions, see *XL C/C++ Compiler Reference* or the options man page.

Table 1. Basic compiler options for optimization

Option	Description
-qnoopt	The compiler performs very limited optimization. This is the default. Before you start optimizing your application, ensure that it compiles successfully with **-qnoopt**.
-O2	The compiler performs comprehensive low-level optimization, which includes graph coloring, common subexpression elimination, dead code elimination, algebraic simplification, constant propagation, instruction scheduling for the target machine, loop unrolling, and software pipelining.
-qarch **-qtune** **-qcache**	The compiler takes advantage of the characteristics of the specific hardware and instruction set where the application run. Use **-qarch** to specify family of processor architectures for which application code should be generated. Use **-qtune** to bias optimization toward execution on a given microprocessor. Use **-qcache** to specify a specific cache or memory geometry.
-qpdf1 **-qpdf2**	When specified with an optimization level of **-O** or higher, the compiler uses profile-directed feedback to optimize the application based on an analysis of how often different sections of code are typically executed. The PDF process is most useful for applications that contain unstructured branching.
-O3	The compiler performs more aggressive optimization than at **-O2**: deeper loop unrolling, better loop scheduling, elimination of the limits on implicit memory usage.
-qhot	The compiler performs high-order transformations, which provide additional loop optimization and optionally performs array padding. This option is most useful for scientific applications that perform a large amount of numerical processing.
-qipa	The compiler performs interprocedural analysis to optimize the entire application as a unit (whole-program analysis). This option is most useful for business applications that contain a large number of frequently used routines. It is also useful for C++ programs with a high level of abstraction. In many cases, this option significantly increases compilation time.
-O4	This is equivalent to **-O3 -qipa -qhot -qarch=auto -qtune=auto -qcache=auto**. If the compilation takes too long, try compiling with **-O4 -qnoipa**.
-O5	This is equivalent to **-O4 -qipa=level=2**. On the Linux platform, this option also turns on **-qhot=vector -qhot=simd**, provided that the processor is PowerPC 970 and that AltiVec data types are supported by the operating system.

Porting considerations

This section describes general areas of investigation that can facilitate porting a UNIX-based application to the Linux platform.

Porting an application to run on another platform involves a *source* platform and a *target* platform. At the most basic level, the following question should be considered before doing any coding: *What is being changed as part of the port to the target platform?* A true port involves changing only the hardware and operating system.

In theory, well-written programs that do not rely on platform-specific dependencies, adhere to industry standards, such as POSIX, and conform to standard language definitions without employing nonstandard language extensions, can easily be ported to a new operating system with a minimum of extra work besides recompiling and debugging. When the source platform is a reasonably recent UNIX-based operating system, the changes may be confined to becoming more compliant with industry standards or with a newer version of the same standard. If the application is already running on a Linux system, you have the option to recompile it and run it natively. Many applications recompile and run without change.

Moreover, compiling an application with different standards-conforming compilers can drive out subtle weaknesses in the source code due to differences in the implementations of the language standards. The result is that the application becomes more robust.

Problems that arise in a porting exercise can be classified into internal and external portability issues. An *internal* portability issue deals with implicit assumptions that are intrinsic to the programming language. For example, C programs assume a particular byte order within integers, a set of relative sizes of integers, and a particular layout of fields within structures. Internal portability pertains to the relationship of the program code to the hardware. This class of porting problems is under a programmer's control.

On the other hand, *external* portability issues pertain to the choice of external interfaces that a program uses, the semantics of these interfaces that are assumed by the program, and the arguments and return values that are passed to and from the program. They deal with libraries and system calls that the program depends upon, code that is invoked by, but external to, the program. External portability can be under a programmer's control if the program uses standardized external interfaces.

Portability issues intrinsic to the language

This section presents some internal portability considerations related to porting to the Linux platform.

- Checking the amount of reliance on GNU C and other language extensions. An application that conforms strictly to its ISO language specification will be maximally portable. IBM XL C/C++ supports a subset of the GNU C and C++ extensions to C and C++. You may need to revisit code that relies on unsupported extensions.
- Checking how null pointers are dereferenced. Some errors in the code can go undetected on a platform due to hardware-dependent characteristics. These

kinds of errors might show up when the program is ported to another platform. If AIX is the source platform, you can use the option **-qcheck=nullptr** to help detect such conditions before porting.

The lowest 4K of memory (that is, addresses 0 through 4K-1) are readable and contain zeroes on AIX, but are not readable on the Linux and Mac OS X platforms, and will cause a segmentation violation if accessed on those platforms. For example,

```
if (strcmp(a, NULL) == 0) ...
```

results in a segmentation violation on Linux and Mac OS X, but not on AIX.

- Changing the C++ object model. Only the GNU C++ object model is supported for the Linux platform. If you are porting an AIX makefile that uses **-qobjmodel** to specify an object model, you will need to delete that option and all other references to the object model.
- Checking the alignment. The types of alignment supported on AIX are not the same as those supported on the Linux or Mac OS X platforms, even though the names might be the same. If you are porting a program that relies on specific values for **-qalign** or `#pragma align`, you may need to change the program.
- Ensuring the portability of data structures. If you generate data with an application on one platform and read the data with an application on another platform, the data may have an alignment that is different from that which the reading application expects. To avoid this problem, make sure that you use a platform-neutral mechanism for the layout of data in structures. For example, if you enclose a structure with `#pragma pack(1)` and `#pragma pack(pop)` pair, the alignment will be the same on all platforms.
- Using the **gxlc** or **gxlc++** utility for translating the commands in your makefiles. Not all gcc or g++ options have an XL C/C++ equivalent.
- Using a different compiler invocation mode for 32-bit or 64-bit applications.
- On Linux or Mac OS X, if the default global `operator new` is called and the allocation request cannot be fulfilled, an exception of type `std::bad_alloc` is thrown. On AIX, the default behavior of the global `operator new` is to return a null pointer if allocation fails.
- Ensuring the portability of applications that use templates. The C++ compiler provides two different methods of working with template files, as alternatives to maintaining templates manually in the source code. Each method has an associated compiler option. The `-qtemplateregistry` compiler option maintains a record of all templates. This method is recommended. An older compiler option, `-qtempinc`, is also provided for applications that you port from another platform. However, on the Mac OS X platform, the compiler option `-qtempinc` is considered deprecated.

Related References

The following IBM Redbooks contain information related to porting. Other Redbooks are available online at www.redbooks.ibm.com.
- *Linux Applications on pSeries* (2003).

Diagnostics for compile-time errors

A basic recommendation for a porting project is to compile the application with the option **-qinfo=por**. This suboption of **-qinfo** adds diagnostic messages that pertain specifically to portability. The option **-qinfo=warn64** instructs the compiler to emit diagnostic messages specific to porting an application to 64-bit mode. These messages can help to narrow the scope of investigation or to pinpoint a particular coding construct.

The following table shows other options that can be helpful for detecting and correcting compile-time errors.

Other diagnostic options for compile-time errors

Option	Description
-qsrcmsg	Prints to standard error the source line that the compiler believes to contain the error, a line below it pointing to a particular spot in that source line, and the diagnostic message.
-qsource	Requests a compiler listing to be returned. The various sections of a listing include the source code with line numbers, the options specified, a listing of all files used in the compilation, a summary of the diagnostic messages by severity level, the number of lines read, and whether or not the compilation was successful. The attribute and cross-reference section of a listing can be produced by specifying the **-qattr** and **-qxref** options, respectively. The object section, which requires specifying the option **-qlist**, shows the pseudo assembly code generated by the compiler and is used for diagnosing execution time problems in cases where you suspect the program is not performing as expected due to code generation errors.
-qsuppress	Stops particular messages from being emitted by the compiler. You can suppress more than one message by listing the message numbers in a colon separated list.
-qflag	Stops specified diagnostic messages from being emitted to the terminal and the listing file. The option uses the single-letter severity codes of the default compiler message format to specify the level below which messages should be ignored.

32-bit and 64-bit application development

You can use XL C/C++ to develop both 32-bit and 64-bit applications. This section contains reference information and other portability considerations for moving C and C++ programs from 32-bit to 64-bit mode.

A porting exercise is a true port if the hardware and operating systems are the only things that are changed. Moving a 32-bit application to a 64-bit programming model as part of the process of migrating to Linux means that the exercise is no longer a true port but a development activity. A 32-bit application with any of the following characteristics is very likely to require changes when ported to a 64-bit environment:
- Reads and interprets kernel memory directly.
- Uses the /proc file system to access 64-bit processes.
- Uses a library that has only a 64-bit version.

- Is a device driver.
- Is being ported from a source platform that has interoperability issues with Linux.

Developing in 64-bit mode allows the application to take advantage of the newer, faster 64-bit hardware and operating systems to improve performance on large, complex, memory-intensive programs, such as database and scientific applications. Applications that are limited by a 32-bit address space, such as database applications, Web search engines, and scientific computing applications, are likely to benefit from a transition to the 64-bit mode. I/O bound applications in 64-bit mode can also realize improved performance by keeping data in memory rather than writing to disk, since disk I/O is usually slower than memory access.

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

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

Other disadvantages of 64-bit applications are that they require more stack space to hold the larger registers. Applications have a bigger cache footprint due to the larger pointer size. 64-bit applications do not run on 32-bit platforms.

Some ways to compensate for the performance liabilities of 64-bit programs are listed below.

- Avoid performing mixed 32-bit and 64-bit operations. For example, adding a 32-bit signed data type to a 64-bit data type requires the 32-bit type to be sign-extended to set the upper 32 bits of the register according to the sign. Setting the upper bits of the register slows the computation. If the 32-bit type were unsigned, then the upper bits of the register would be cleared.
- Avoid 64-bit long division whenever possible. Multiplication is usually faster than division. If you need to perform many divisions with the same divisor, assign the reciprocal of the divisor to a temporary variable, and change all divisions to multiplications with the temporary variable. For example, the function

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

will perform faster than the more straightforward:

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

The reason is that the division (`1.0 / 1.0825`) is evaluated once at compile time only, due to constant folding. This optimization is usually done by the compiler if the **-qnostrict** option is in effect (true when compiling at optimization levels **-O2** and higher).

- Use **long** variables as array indexes instead of signed, unsigned, and plain **int** types. Doing so frees the compiler from having to perform sign extension during array references.

Well-written code is likely to compile and run correctly with a minimum of rework and debugging if moved to the 64-bit programming model. The term *well-written* implies conformance to the language standard and adherence to good programming

practices. In the context of moving to a 64-bit programming model, the term also includes the notions that the code does not depend on a specific byte order nor on external data formats, and that it uses function prototypes, appropriate system header files, and system-derived data types

Diagnostics for run-time errors

A program might compile and link successfully, yet produce unexpected results during execution. The compiler does not diagnose programming errors that do not violate the syntax of the language. This section describes some common errors, how to detect them, and how to correct them.

Uninitialized variables

An object of automatic storage duration is not implicitly initialized and its initial value is therefore indeterminate. If an **auto** variable is used before it is set, it may or may not produce the same results every time the program is run. The **-qinfo=gen** compiler option displays the location of **auto** variables that are used before being set. The **-qinitauto** option instructs the compiler to initialize all automatic variables to the specified value. This option reduces the run-time performance of the application and is recommended for debugging only.

Run-time checking

The **-qcheck** option inserts run-time checking code into the executable. The suboptions specify checking for null pointers, indexing outside of array bounds, and division by zero. Like **-qinitauto**, **-qcheck** degrades application performance and is recommended for debugging purposes only.

ANSI aliasing

Type-based aliasing, also referred to as ANSI aliasing, restricts the lvalues that can be safely used to access a data object. When compiling under a language level that enforces conformance to the language standards, the C and C++ compilers enforce type-based aliasing during optimization. The ANSI aliasing rule states that a pointer can only be dereferenced to an object of the same type or a compatible type. The exceptions are that sign and type qualifiers are not subject to type-based aliasing and that a character pointer can point to any type. The common coding practice of casting a pointer to an incompatible type and then dereferencing it violates this rule.

Turning off ANSI aliasing by setting **-qalias=noansi** may correct program behavior, but doing so reduces the opportunities for the compiler to optimize the application and thereby degrades run-time performance. The recommended solution is to change the program to conform to the type-based aliasing rule.

#pragma option_override

Sometimes an error appears only when optimization is used. It can be worthwhile, especially for complex applications, to turn off optimization for a function known to contain a programming error, while allowing the rest of the program to be optimized. The `#pragma option_override` directive allows you to specify alternate optimization options for specific functions.

The `#pragma option_override` directive can also be used to determine the function is causing the problem. The discovery is made by selectively turning off optimization for each function within the directive until the problem disappears.

Shared memory parallelization

Several proven techniques are available to achieve parallel execution of a program and more rapid job completion than running on a single processor. These techniques include:

- Directive-based shared memory parallelization (SMP)
- 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()`

The portability requirements for the application are definitely a factor in selecting the best technique to use. The choice is also highly dependent on the application, the programmer's skills and preferences, and the characteristics of the target machine. The two principal ways of accomplishing parallelization on AIX is by using hand-coded POSIX threads (pthreads) and OpenMP directives.

The parallel programming facilities of the AIX operating system are based on the concept of threads. Parallel programming exploits the advantages of multiprocessor systems, while maintaining a full binary compatibility with existing uniprocessor systems. This means that a multithreaded program that works on a uniprocessor system can take advantage of a multiprocessor system without recompiling.

Pthreads offer great flexibility for making effective use of multiple processors because they provide the maximal amount of control of the parallelization process. The trade-off is a considerable increase in code complexity. In many cases, the simpler approach of using OpenMP directives, SMP-enabled libraries, or the automatic SMP capabilities of compilers is preferable. The explicit use of threads does not necessarily lead to better performance. Debugging multithreaded applications is awkward at best. However, in some programs, it is the only viable approach.

The following lists some pros and cons of the different techniques.

Automatic parallelization by the compiler
- Easy to implement (compile with `-qsmp=auto`).
- Enables teamwork easily.
- Limited scalability because data scoping is neglected.
- Compiler-dependent (even on the release of a particular compiler).
- Not necessarily portable.

SMP-enabled libraries
- Requires the least effort.
- Not necessarily portable (usually is proprietary).
- Limited flexibility.

OpenMP directives
- Portable.
- Potentially better scalability of the automatic parallelization.
- Uniform memory access is assumed.

Hybrid approach (subset or mixture of OpenMP and pthreads, or UNIX `fork()` and `exec()` parallelization, platform-specific constructs)
- Might enable teamwork.
- Needs a well-tested concept to assure performance and portability.
- Not necessarily portable.

pthreads
- Portable.
- Provides maximal control over the parallelization process.
- Potentially the best scalability of the automatic parallelization.
- Needs experienced programmers to handle code complexity.

OpenMP directives

OpenMP directives are a set of commands that instruct the compiler how a particular loop should be parallelized. The existence of the directives in the source removes the need for the compiler to perform any parallel analysis on the parallel code. The use of OpenMP directives requires the presence of the pthread libraries to provide the necessary infrastructure for parallelization.

The OpenMP directives address three important issues of parallelizing an application. First, 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. Second, work sharing directives specify how the work contained in a parallel region of code should be distributed across the SMP processors. Finally, there are directives for synchronization between the processors.

The compiler supports the OpenMP Version 2.0 specification.

Related References
- Appendix B, "OpenMP compliance and support," on page 51

Features related to GNU C and C++ portability

To facilitate porting an application or code developed with GNU C, XL C/C++ supports a subset of the GNU C and C++ language extensions to C99 and Standard C++. The tables in this section list the features that are supported, unsupported, and those for which the syntax is accepted but the semantics ignored.

To use *supported* extensions with your C code, use the **xlc** or **cc** invocation commands, or specify one of `-qlanglvl=extc89`, `-qlanglvl=extc99`, or `-qlanglvl=extended`. In C++, all supported GNU C and C++ features are accepted by default.

In the following tables, extensions marked *accept/ignore* are recognized by the compiler as acceptable programming keywords, but the GNU C/C++ semantics are not supported. This means that compilation does not halt if the compiler encounters an *accept/ignore* keyword or extension, but the GNU semantics are not implemented in the application. Compiling source code that uses these extensions under a strict language level (`stdc89`, `stdc99`) will result in error messages.

Related References

The GNU C and C++ language extensions are fully documented in the GNU manuals at http://gcc.gnu.org/onlinedocs.

Function attributes

Use the keyword __**attribute**__ to specify special attributes when making a function declaration or definition. This keyword is followed by an attribute specification inside double parentheses. XL C/C++ supports a subset of the GNU C and C++ function attributes. Behavior described as *accept/ignore* means that the syntax is accepted, but the semantics are ignored, and compilation continues.

GNU C/C++ function attribute compatibility with XL C/C++

Function Attribute	Behavior
alias	supported
always_inline	supported
cdecl	accept/ignore
const	supported
constructor	supported
destructor	supported
dllexport	accept/ignore
dllimport	accept/ignore
eightbit_data	accept/ignore
exception	accept/ignore
format	supported
format_arg	supported
function_vector	accept/ignore
interrupt	accept/ignore
interrupt_handler	accept/ignore
longcall	accept/ignore
model	accept/ignore
no_check_memory_usage	accept/ignore
no_instrument_function	accept/ignore
noinline	supported
noreturn	supported
pure	supported
regparm	accept/ignore
section	supported
stdcall	accept/ignore
tiny_data	accept/ignore
weak	supported

Related References
- ″Function attributes″ in *XL C/C++ Language Reference*

Variable attributes

Use the keyword __**attribute**__ to specify special attributes of variables or structure fields. This keyword is followed by an attribute specification inside double parentheses. XL C/C++ supports a subset of the GNU C and C++ variable attributes. Behavior described as *accept/ignore* means that the syntax is accepted, but the semantics are ignored, and compilation continues.

GNU C/C++ variable attribute compatibility with XL C/C++

Variable Attribute	Behavior
aligned	supported
▶ C++ init_priority	supported
mode	supported
model	accept/ignore
nocommon	supported
packed	supported
section	supported
transparent_union	supported
unused	accept/ignore
weak	supported

Related References
• ″Variable attributes″ in *XL C/C++ Language Reference*

Type attributes

Use the keyword __**attribute**__ to specify special attributes of `struct` and `union` types when you define these types. This keyword is followed by an attribute specification inside double parentheses. XL C/C++ supports a subset of the GNU C and C++ type attributes. Behavior described as *accept/ignore* means that the syntax is accepted, but the semantics are ignored, and compilation continues.

GNU C/C++ type attribute compatibility with XL C/C++

Type Attribute	Behavior
aligned	supported
packed	supported
transparent_union	supported
unused	accept/ignore

Related References
• ″Type attributes″ in *XL C/C++ Language Reference*

GNU C and C++ assertions

Use *assertions* to test what sort of computer or system the compiled program will run on. The assertions `#cpu`, `#machine`, and `#system` are predefined. You can also define assertions with the preprocessing directives `#assert` and `#unassert`.

GNU C Assertions	Behavior
`#assert`	supported
`#unassert`	supported
`#cpu`	supported possible value is `powerpc`
`#machine`	supported possible values are `powerpc` and `bigendian`
`#system`	supported possible values are `unix` and `posix`

Other GNU C and C++ portability issues

The following features related to GNU C and C++ are supported under extended language levels (`extc89`, `extc99`, `extended`).

- Use directive `#warning` to cause the preprocessor to issue a warning and continue processing.
- Use directive `#include_next` to specify inclusion of the next header file in a directory after the current one.
- Local labels can be declared at the start of each statement expression.
- Use a brace-enclosed compound statement inside of parentheses as an expression.
- Refer to the type of an expression with the **__typeof__** keyword.
- Use compound expressions, conditional expressions, and casts as lvalues.
- Use a computed **goto** statement to jump to a label, which has had its address taken and the address is used as a value.
- Use keyword **__alignof__** to inquire about variable alignment, or the alignment usually required by a type.
- Use alternate spelling of these keywords: **__asm__**, **__const__**, **__volatile__**, **__inline__**, **__signed__**, and **__typeof__**.
- Use the **__extension__** keyword to avoid an error when using an extended language feature in a strict language level mode.
- An array of zero length can occur without generating an error.
- Allow forward declarations of template instantiations by using the **extern** keyword.
- ▶ `C` A function definition that appears within the definition of another function (a nested function) is permitted.
- ▶ `C` A union member can be cast to the union type to which it belongs.

Under extended language levels (`extc89`, `extc99`, `extended`), XL C/C++ recognizes the syntax of the following features, but their semantics are not supported.

- ▶ `C++` The declaration of a register variable, either global or local, can suggest a preferred register.

Standard C++ Numerics Library header files

It should be noted that not all of the math functions in the XL C++ header files `<cmath>` and `<cstdlib>` can handle an argument of integral type. The reason is that the conversion of an argument from an integral type to a floating-point type creates multiple overload resolution possibilities of equal rank.

For example, taking the absolute value of an **unsigned int** or an **unsigned long** in a C++ program results in an overload resolution error, despite the fact that the XL C++ implements all of the required signatures for the `abs()` library function in its headers.

```
int abs(int);
long abs(long);
float abs(float);
double abs(double);
long double abs(long double);
```

The problem persists even if both of the XL headers `<math.h>` and `<stdlib.h>` are included. The workaround is to cast an argument of unsigned integral type to a suitable type before using it in the function.

Appendix A. Language support

This appendix discusses the implementations of the C and C++ programming languages and the language extensions provided by XL C/C++.

Compatibility with ISO/IEC International Standards

XL C/C++ can foster a programming style that emphasizes portability. Syntax and semantics constitute a complete specification of a programming language, but conforming implementations of a particular language specification can differ due to language extensions.

A program that conforms strictly to its language specification will have maximum portability among different environments. In theory, a program that compiles correctly with one standards-conforming compiler and that does not use any extension or implementation-defined behavior, will compile and execute properly under all other conforming compilers, insofar as hardware differences permit. A program that correctly exploits the extensions to the language that are provided by the language implementation can improve the efficiency of its object code.

ISO/IEC 14882:2003(E) International Standard compatibility

XL C/C++ is consistent with the ISO/IEC International Standard 14882:2003(E), which specifies the form and establishes the interpretation of programs written in the C++ programming language. The international standard is designed to promote the portability of C++ programs among a variety of implementations. ISO/IEC 14882:1998 was the first C++ language.

ISO/IEC 9899:1990 International Standard compatibility

The ISO/IEC 9899:1990 International Standard (also known as C89) specifies the form and establishes the interpretation of programs written in the C programming language. This specification is designed to promote the portability of C programs among a variety of implementations. This Standard was amended and corrected by ISO/IEC 9899/COR1:1994, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996. To ensure that your source code adheres strictly to the amended and corrected C89 standard, specify the -qlanglvl=stdc89 compiler option.

ISO/IEC 9899:1999 International Standard support

The ISO/IEC 9899:1999 International Standard (also known as C99) is an updated standard for programs written in the C programming language. It is designed to enhance the capability of the C language, provide clarifications to C89, and incorporate technical corrections. XL C/C++ supports many features of this language specification.

▶ C ◀ The C compiler supports all language features specified in the C99 Standard. To ensure that your source code adheres to this set of language features, use the **c99** invocation command. Note that the Standard also specifies features in the run-time library. These features may not be supported in the current run-time library and operating environment. The availability of system header files provides an indication of whether such support exists.

Major features in C99
XL C/C++ implements all C99 language features. The following is a table of selected major features. The references refer to articles in *XL C/C++ Language*

47

Reference.

ISO/IEC 9899:1999 international standard extensions to IBM C

C99 Feature	Related Reference
`restrict` type qualifier for pointers	The `restrict` Type Qualifier
universal character names	The Unicode Standard
predefined identifier `__func__`	Predefined Identifiers
function-like macros with variable and empty arguments	Function-Like Macros
`_Pragma` unary operator	The _Pragma Operator
variable length array	Variable Length Arrays
`static` keyword in array index declaration	Arrays
`complex` data type	Complex Types
`long long int` and `unsigned long long int` types	Integer Variables
hexadecimal floating-point constants	Hexadecimal Floating Constants
compound literals for aggregate types	Compound Literals
designated initializers	Initializers
C++ style comments	Comments
implicit function declaration not permitted	Function Declarations
mixed declarations and code	The for Statement
`_Bool` type	Simple Type Specifiers
`inline` function declarations	Inline Functions
initializers for aggregates	Initializing Arrays Using Designated Initializers

Changes and clarifications of C89 supported in C99

Certain specifications in the C99 Standard are based on changes and clarifications of the C89 standard, rather than on new features of the language. XL C/C++ supports all C99 language features, including the following:

- Flexible array members are allowed. The last member of a structure with two or more members can be declared without the size.
- Declaring implicit `int` is not supported. All declarations must have a type specifier.
- Trailing commas are allowed in enumeration specifiers.
- Duplicate type qualifiers are accepted and ignored, unless explicitly specified otherwise.
- A diagnostic message will be issued if a required expression is missing from the return statement.
- Constant expressions evaluated during preprocessing now use `long long` and `unsigned long long` data types.
- Empty macro arguments are allowed in function-like macros.
- The maximum value of `#line` has increased to `2 147 483 647`.

C99 features in XL C/C++

Some features of the ISO/IEC 9899:1999 International Standard (C99) are also implemented in C++. These extensions are available under the `-qlanglvl=extended` compiler option.

ISO/IEC 9899:1999 international standard extensions to IBM C++

C99 Feature	Reference
`restrict` type qualifier for pointers	The `restrict` Type Qualifier
universal character names	The Unicode Standard
predefined identifier `__func__`	Predefined Identifiers
variable length array	Variable Length Arrays
`complex` data type	Complex Types
hexadecimal floating-point constants	Hexadecimal Floating Constants
compound literals for aggregate types	Compound Literals
function-like macros with variable and empty arguments	Function-Like Macros
`_Pragma` unary operator	The _Pragma Operator

Enhanced language level support

The `-qlanglvl` compiler option is used to specify the supported language level, and therefore affects the way your code is compiled. You can also specify the language level implicitly by using different compiler invocation commands. In general, a valid program that compiles and runs correctly under a standard language level should continue to compile correctly and run to produce the same result with the orthogonal extensions enabled.

For example, to compile C programs so that they comply strictly with the ISO/IEC 9899:1990 International Standard (C89), you need to specify `-qlanglvl=stdc89`. The `stdc89` suboption instructs the compiler to strictly enforce the standard, and not to allow any language extensions. (The **c89** compiler invocation command specifies this language level implicitly.)

You can also use extensions to the standard language levels. Extensions that do not interfere with the standard features are called *orthogonal* extensions. For example, when you compile C programs, you can enable extensions that are orthogonal to C89 by specifying `-qlanglvl=extc89`.

Most of the language features described in the ISO/IEC 9899:1999 International Standard (C99) are considered orthogonal extensions to C89.

Non-orthogonal extensions, on the other hand, can interfere or conflict with aspects of the language as described in one of the international standards. Acceptance of these extensions must be explicitly enabled by a particular compiler option. Reliance on non-orthogonal extensions reduces the ease with which your application can be ported to different environments.

The main suboptions for the `-qlanglvl` option are listed below.

Selected -qlanglvl suboptions

-qlanglvl Suboption	Suboption Description
`-qlanglvl=stdc99`	Specifies strict conformance to the C99 standard.
`-qlanglvl=stdc89`	Specifies strict conformance to the C89 standard.
`-qlanglvl=extc99`	Enables all extensions orthogonal to C99.
`-qlanglvl=extc89`	Enables all extensions orthogonal to C89.

Selected -qlanglvl suboptions

-qlanglvl Suboption	Suboption Description
-qlanglvl=extended	Enables all extensions orthogonal to C89 and specifies the -qupconv compiler option.

Appendix B. OpenMP compliance and support

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

XL C/C++ is compliant with OpenMP Specification 2.0. The compiler recognizes and preserves the semantics of the following OpenMP V2.0 elements:
- Comma delimiter for multiple clauses in the `#pragma omp` directive.
- The `num_threads` clause.
- The `copyprivate` clause.
- `threadprivate` static block scope variables.
- Support for C99 variable length arrays.
- Redundant declaration of private variables.
- Timing routines `omp_get_wtick` and `omp_get_wtime`.

The directives, library functions, and environment variables described below allow you to create and manage parallel programs while maintaining portability.

To enable OpenMP parallel processing, you must specify the **-qsmp** compiler option.
- To choose automated parallelism, specify **-qsmp** or **-qsmp=auto**. This suboption enables the compiler to perform *implicit parallelism*, in addition to recognizing and implementing any OpenMP directives, library functions, and environment variables included in the program.
- To choose strict compliance to the OpenMP Specification 2.0, specify **-qsmp=omp**. This suboption ensures that the compiler implements only the OpenMP directives, library functions, and environment variables specified in the code. It does not perform any additional automated parallel processing.

Related References
- http://www.openmp.org
- "Pragmas to control parallel processing" in *XL C/C++ Compiler Reference*
- "Program parallelization" in *XL C/C++ Compiler Reference*

OpenMP directives

Each directive starts with `#pragma omp`, to reduce the potential for conflict with other pragma directives.

OpenMP directives in XL C/C++

Directive name	Directive Description
parallel	The `parallel` directive defines a *parallel region*, which is a region of the program that is to be executed by multiple threads in parallel.
for	The `for` directive identifies an iterative work-sharing construct that specifies a region in which the iterations of the associated loop should be executed in parallel. The iterations of the `for` loop are distributed across threads that already exist.

OpenMP directives in XL C/C++

Directive name	Directive Description
sections	The `sections` directive identifies a non-iterative work-sharing construct that specifies a set of constructs that are to be divided among threads in a team. Each section is executed once by a thread in the team.
single	The `single` directive identifies a construct that specifies that the associated structured block is executed by only one thread in the team (not necessarily the master thread).
parallel for	The `parallel for` directive is a shortcut form for a parallel region that contains a single `for` directive. The semantics are identical to explicitly specifying a `parallel` directive immediately followed by a `for` directive.
parallel sections	The `parallel sections` directive provides a shortcut form for specifying a parallel region containing a single `sections` directive. The semantics are identical to explicitly specifying a `parallel` directive immediately followed by a `sections` directive.
master	The `master` directive identifies a construct that specifies a structured block that is executed by the master thread of the team.
critical	The `critical` directive identifies a construct that restricts execution of the associated structured block to a single thread at a time. An optional *name* may be used to identify the critical region. A thread waits at the beginning of a critical region until no other thread is executing a critical region with the same name. All unnamed critical directives map to the same unspecified name.
barrier	The `barrier` directive synchronizes all the threads in a team. When encountered, each thread waits until all of the others have reached this point. After all threads have encountered the barrier, each thread begins executing the statements after the barrier directive in parallel.
atomic	The `atomic` directive identifies a specific memory location that must be updated atomically and not be exposed to multiple, simultaneous writing threads.
flush	The `flush` directive identifies a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory.
ordered	The `ordered` directive identifies a structured block of code that must be executed in sequential order.
threadprivate	The `threadprivate` directive declares file-scope, namespace-scope, or static block-scope variables to be private to a thread.

OpenMP data scope attribute clauses

Clauses may be specified on the directives to control the scope attributes of variables for the duration of the parallel or work-sharing constructs.

OpenMP data scope attribute clauses in XL C/C++

Data Scope Attribute Clause Name	Data Scope Attribute Clause Description
`private`	The `private` clause declares the variables in the list to be private to each thread in a team.
`firstprivate`	The `firstprivate` clause provides a superset of the functionality provided by the `private` clause.
`lastprivate`	The `lastprivate` clause provides a superset of the functionality provided by the `private` clause.
`copyprivate`	The `copyprivate` clause provides an alternative to using a shared variable to broadcast a value to a team. The mechanism uses a private variable to broadcast a value from one team member to other members.
`num_threads`	The `num_threads` clause provides the ability to request a specific number of threads for a parallel construct.
`shared`	The `shared` clause shares variables that appear in the list among all the threads in a team. All threads within a team access the same storage area for `shared` variables.
`reduction`	The `reduction` clause performs a reduction on the scalar variables that appear in list, with a specified operator.
`default`	The `default` clause allows the user to affect the data scope attributes of variables.

OpenMP library functions

OpenMP runtime library functions are included in the header `<omp.h>`. They include *execution environment functions* that can be used to control and query the parallel execution environment, and *lock functions* that can be used to synchronize access to data.

OpenMP runtime library functions in XL C/C++

Runtime Library Function Name	Runtime Library Function Description
`omp_set_num_threads`	Sets the number of threads to use for subsequent parallel regions.
`omp_get_num_threads`	Returns the number of threads currently in the team executing the parallel region from which it is called.
`omp_get_max_threads`	Returns the maximum value that can be returned by calls to omp_get_num_threads.
`omp_get_thread_num`	Returns the thread number, within its team, of the thread executing the function. The master thread of the team is thread 0.
`omp_get_num_procs`	Returns the maximum number of processors that could be assigned to the program.
`omp_in_parallel`	Returns non-zero if it is called within the dynamic extent of a parallel region executing in parallel; otherwise, it returns 0.

OpenMP runtime library functions in XL C/C++

Runtime Library Function Name	Runtime Library Function Description
omp_set_dynamic	Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.
omp_get_dynamic	Returns non-zero if dynamic thread adjustment is enabled and returns 0 otherwise.
omp_set_nested	Enables or disables nested parallelism.
omp_get_nested	Returns non-zero if nested parallelism is enabled and 0 if it is disabled.
omp_init_lock	Initializes a simple lock.
omp_destroy_lock	Removes a simple lock.
omp_set_lock	Waits until a simple lock is available.
omp_unset_lock	Releases a simple lock.
omp_test_lock	Tests a simple lock.
omp_init_nest_lock	Initializes a nestable lock.
omp_destroy_nest_lock	Removes a nestable lock.
omp_set_nest_lock	Waits until a nestable lock is available.
omp_unset_nest_lock	Releases a nestable lock.
omp_test_nest_lock	Tests a nestable lock.
omp_get_wtick	Returns the number of seconds between successive clock ticks.
omp_get_wtime	Returns the elapsed wall-clock time in seconds.

OpenMP environment variables

OpenMP environment variables control the execution of parallel code. The names of environment variables must always be in upper case, while their values are not case-sensitive.

Description	Syntax

OMP_SCHEDULE
 	
Sets the run-time schedule type and chunk size. Applies only to OpenMP directives that have the scheduling type set to `runtime`.
	►►—OMP_SCHEDULE=—┬─static──┬─┬──────┬─┬─►◄
	                  │ │ │ ,—*n* │ │
	                  ├─affinity─┤ └──────┘ │
	                  ├─dynamic─┤ ┌──────┐ │
	                  ├─guided──┤ └─,—*n*─┘ │
	                  └─runtime─┘
	
	where
	
	affinity
	   An IBM extension valid for C only. Specifies that iterations of a loop are initially divided into local partitions of a size equal to the ceiling of the number of iterations divided by the number of threads: `CEILING(number_of_iterations ÷ number_of_threads)`. Each local partition is further subdivided into chunks of a size equal to the ceiling of half of the number of iterations remaining in the local partition: `CEILING(iterations_left_in_local_partition ÷ 2)`. When a thread becomes free, it takes the next chunk from its local partition. If no chunks are in the local partition, the thread takes an available chunk from a partition of another thread. If *n* is specified, each local partition is subdivided into chunks of size *n*. If *n* is not specified, the default value is 1.
	
	dynamic
	   Specifies that iterations for a **for** loop should be divided into a series of chunks of size *n* and that the chunks are handled according to the following process. A thread waiting for an assignment is assigned a chunk of iterations, which it executes and then waits for its next assignment. This process is repeated until all chunks are assigned. If *n* is not specified, the default chunk size is 1.
	
	guided
	   Specifies that iterations for a **for** loop should be assigned to threads in chunks with decreasing sizes and that the chunks are handled according to the following process. A thread that finishes its assigned chunk of iterations is dynamically assigned another chunk, until all chunks are assigned. If *n* is not specified, the default value for the initial chunk size is 1.
	
	static
	   Specifies that iterations for a **for** loop should be divided into a series of chunks of size *n* and that the chunks are handled according to the following process. Available threads are assigned chunks in an order determined by the thread number. When *n* is not specified, the iteration space is divided into chunks that are approximately equal in size, with one chunk assigned to each thread.
	
	n Is a positive number, representing the chunk size.

OpenMP environment variables in XL C/C++

Description	Syntax
OMP_DYNAMIC Enables or disables dynamic adjustment of the number of threads available for the execution of parallel regions.	►►──OMP_DYNAMIC=──┬─true──┬──────────►◄ 　　　　　　　　　　　　└─false─┘ where **true** 　　Enables dynamic adjustment of the number of threads available. **false** 　　Disables dynamic adjustment of the number of threads available.
OMP_NUM_THREADS Sets of the number of threads available for the execution.	►►──OMP_NUM_THREADS=*n*──────────►◄ where *n*　Represents the number of threads.
OMP_NESTED Enables or disables nested parallelism.	►►──OMP_NESTED=──┬─true──┬──────────►◄ 　　　　　　　　　　　└─false─┘ where **true** 　　Enables nested parallelism. **false** 　　Disables nested parallelism.

OpenMP implementation-defined behavior

The following information is not specified in the standard. Each implementation of the standard may have its own implementation-defined values.

Conditional Compilation
The `_OPENMP` macro is defined to 199810.

Scheduling　The `schedule` clause specifies how iterations of a **for** loop are divided among threads of the team. The possible OpenMP standard values are `static`, `dynamic`, `guided`, and `runtime`. In addition, IBM C adds the value `affinity` as an extension. In the absence of an explicitly defined `schedule` clause, the default schedule for XL C/C++ is `static`.

Notices

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

COPYRIGHT LICENSE:

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

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, 2005. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

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 obtains 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.

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

Trademarks and service marks

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

AIX	PowerPC
@server	PowerPC Architecture
IBM	pSeries
POWER3	Redbooks
POWER4	VisualAge
POWER5	

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

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

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

Industry standards

The following standards are supported:
- The C language is consistent with the International Standard C (ANSI/ISO-IEC 9899–1990 [1992]). This standard has officially replaced American National Standard for Information Systems-Programming Language C (X3.159–1989) and is technically equivalent to the ANSI C standard. The compiler supports the changes adopted into the C Standard by ISO/IEC 9899:1990/Amendment 1:1994.
- The C language is consistent with the International Standard for Information Systems-Programming Language C (ISO/IEC 9899–1999 (E)).
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998), the first formal definition of the language.
- The C++ language is also consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:2003 (E)), currently referred to as *Standard C++*.
- The C and C++ compilers support the OpenMP C and C++ Application Programming Interface Version 2.0.

IBM ®

Program Number: 5724-K77