IBM XL C/C++ for Linux, V11.1

IBM

Compiler Reference

Version 11.1

IBM XL C/C++ for Linux, V11.1

Compiler Reference

Version 11.1

First edition

This edition applies to IBM XL C/C++ for Linux, V11.1 (Program 5724-X14) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Contents

Chapter 5. Compiler pragmas reference 329

Chapter 6. Compiler predefined macros 391

Chapter 7. Compiler built-in functions 405

About this information

This information is a reference for the IBM® XL C/C++ for Linux®, V11.1 compiler. Although it provides information on compiling and linking applications written in C and C++, it is primarily intended as a reference for compiler command-line options, pragma directives, predefined macros, built-in functions, environment variables, and error messages and return codes.

Who should read this information

This information is for experienced C or C++ developers who have some familiarity with the XL C/C++ compilers or other command-line compilers on UNIX® operating systems. It assumes thorough knowledge of the C or C++ programming language, and basic knowledge of operating system commands. Although this information is intended as a reference guide, programmers new to XL C/C++ can still find information in it on the capabilities and features unique to the XL C/C++ compiler.

How to use this information

Unless indicated otherwise, all of the text in this reference pertains to both C and C++ languages. Where there are differences between languages, these are indicated through qualifying text and icons, as described in "Conventions" on page viii.

Throughout this information, the **xlc** and **xlc++** command invocations are used to describe the actions of the compiler. You can, however, substitute other forms of the compiler invocation command if your particular environment requires it, and compiler option usage will remain the same unless otherwise specified.

While this information covers topics on configuring the compiler environment, and compiling and linking C or C++ applications using XL C/C++ compiler, it does not include the following topics:

* Compiler installation: see the *XL C/C++ Installation Guide.*for information on installing XL C/C++.
* The C or C++ programming languages: see the *XL C/C++ Language Reference* for information on the syntax, semantics, and IBM implementation of the C or C++ programming languages.
* Programming topics: see the *XL C/C++ Optimization and Programming Guide* for detailed information on developing applications with XL C/C++, with a focus on program portability and optimization.

How this information is organized

Chapter 1, "Compiling and linking applications," on page 1 discusses topics related to compilation tasks, including invoking the compiler, preprocessor, and linker; types of input and output files; different methods for setting include file path names and directory search sequences; different methods for specifying compiler options and resolving conflicting compiler options; how to reuse GNU C/C++ compiler options through the use of the compiler utilities **gxlc** and **gxlc++**; and compiler listings and messages.

Chapter 2, "Configuring compiler defaults," on page 23 discusses topics related to setting up default compilation settings, including setting environment variables, customizing the configuration file, and customizing the **gxlc** and **gxlc++** option mappings.

Chapter 4, "Compiler options reference," on page 67 begins with a summary of options according to functional category, which allows you to look up and link to options by function; and includes individual descriptions of each compiler option sorted alphabetically.

Chapter 5, "Compiler pragmas reference," on page 329 begins with a summary of pragma directives according to functional category, which allows you to look up and link to pragmas by function; and includes individual descriptions of pragmas sorted alphabetically, including OpenMP directives.

Chapter 6, "Compiler predefined macros," on page 391 provides a list of compiler macros according to category.

Chapter 7, "Compiler built-in functions," on page 405 contains individual descriptions of XL C/C++built-in functions for Power architectures, categorized by their functionality.

Conventions

Typographical conventions

The following table explains the typographical conventions used in the IBM XL C/C++ for Linux, V11.1 information.

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Lowercase commands, executable names, compiler options, and directives.	The compiler provides basic invocation commands, **xlc** and **xlC** (**xlc++**), along with several other compiler invocation commands to support various C/C++ language levels and compilation environments.
italics	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	Make sure that you update the *size* parameter if you return more than the *size* requested.
underlining	The default setting of a parameter of a compiler option or directive.	nomaf | <u>maf</u>
`monospace`	Programming keywords and library functions, compiler builtins, examples of program code, command strings, or user-defined names.	To compile and optimize myprogram.c, enter: `xlc myprogram.c -03`.

Qualifying elements (icons)

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

Table 2. Qualifying elements

Qualifier/Icon	Meaning
C only, or C only begins `► C` `C ◄` C only ends	The text describes a feature that is supported in the C language only; or describes behavior that is specific to the C language.
C++ only, or C++ only begins `► C++` `C++ ◄` C++ only ends	The text describes a feature that is supported in the C++ language only; or describes behavior that is specific to the C++ language.
IBM extension begins `► IBM` `IBM ◄` IBM extension ends	The text describes a feature that is an IBM extension to the standard language specifications.
C++0x, or C++0x begins `► C++0x` `C++0x ◄` C++0x ends	The text describes a feature that is introduced into standard C++ as part of C++0x.

Syntax diagrams

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

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

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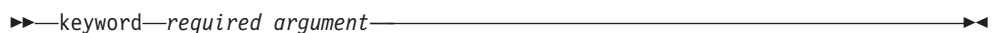
  The ──► symbol indicates that the command, directive, or statement syntax is continued on the next line.

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

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

  Fragments, which are diagrams of syntactical units other than complete commands, directives, or statements, start with the |── symbol and end with the ──| symbol.

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

  ►►──keyword──*required_argument*──►◄

- Optional items are shown below the main path:

```
►►──keyword─────────────────────────────────────────────────────►◄
              └─optional_argument─┘
```
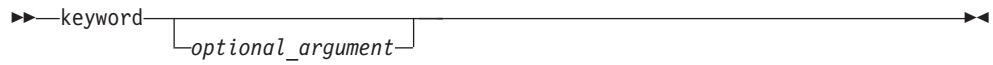
- If you can choose from two or more items, they are shown vertically, in a stack.
  If you *must* choose one of the items, one item of the stack is shown on the main path.

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

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

```
►►──keyword─────────────────────────────────────────────────────►◄
             ├─optional_argument1─┤
             └─optional_argument2─┘
```
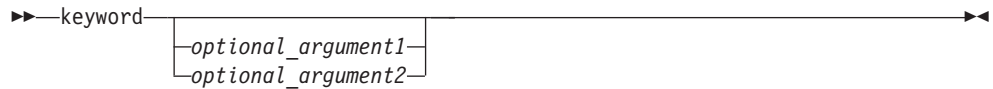
- An arrow returning to the left above the main line (a repeat arrow) indicates that you can make more than one choice from the stacked items or repeat an item. The separator character, if it is other than a blank, is also indicated:

```
             ┌─,──────────────────┐
             ▼                     │
►►──keyword────repeatable_argument─┴─────────────────────────────►◄
```

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

```
              ┌─default_argument───┐
►►──keyword───┴─alternate_argument─┴─────────────────────────────►◄
```

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

Sample syntax diagram

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

```
     (1)    (2)      (3)         (4)    (5)                                    (9)  (10)
►►──────#──────pragma──────comment──────(──┬─compiler─────────────────┬──)──────────►◄
                                           ├─date─────────────────────┤
                                           ├─timestamp────────────────┤
                                           │                 (6)      │
                                           ├─copyright────────────────┤
                                           └─user──┬──────────────────┤
                                                   │    (7)       (8) │
                                                   └─,──"─token_sequence─"─┘
```
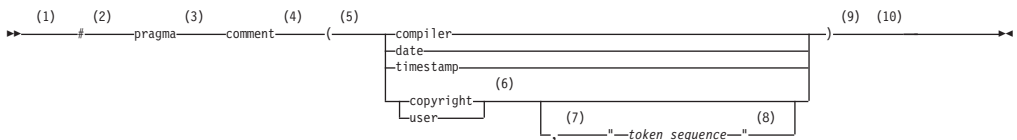
Notes:

1   This is the start of the syntax diagram.

2   The symbol # must appear first.

3   The keyword pragma must appear following the # symbol.

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

5   An opening parenthesis must be present.

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

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

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

9   A closing parenthesis is required.

10   This is the end of the syntax diagram.

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

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

Examples in this information

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

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

Related information

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

IBM XL C/C++ information

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

- README files

    README files contain late-breaking information, including changes and
    corrections to the product information. README files are located by default in
    the XL C/C++ directory and in the root directory of the installation CD.

- Installable man pages

    Man pages are provided for the compiler invocations and all command-line
    utilities provided with the product. Instructions for installing and accessing the
    man pages are provided in the *IBM XL C/C++ for Linux, V11.1 Installation Guide*.

- Information center

    The information center of searchable HTML files can be launched on a network
    and accessed remotely or locally. Instructions for installing and accessing the
    online information center are provided in the *IBM XL C/C++ for Linux, V11.1
    Installation Guide*.

    The information center is viewable on the Web at http://
    publib.boulder.ibm.com/infocenter/lnxpcomp/v111v131/index.jsp.

- PDF documents

PDF documents are located by default in the /opt/ibmcmp/vacpp/11.1/doc/
LANG/pdf/ directory, where *LANG* is one of en_US, zh_CN, or ja_JP. The PDF
files are also available on the Web at http://www.ibm.com/software/awdtools/
xlcpp/linux/library/.

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

Table 3. XL C/C++ PDF files

Document title	PDF file name	Description
IBM XL C/C++ for Linux, V11.1 Installation Guide, GI11-7914-00	install.pdf	Contains information for installing XL C/C++ and configuring your environment for basic compilation and program execution.
Getting Started with IBM XL C/C++ for Linux, V11.1, GI11-7913-00	getstart.pdf	Contains an introduction to the XL C/C++ product, with information on setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
IBM XL C/C++ for Linux, V11.1 Compiler Reference, SC23-8606-00	compiler.pdf	Contains information about the various compiler options, pragmas, macros, environment variables, and built-in functions, including those used for parallel processing.
IBM XL C/C++ for Linux, V11.1 Language Reference, SC23-8607-00	langref.pdf	Contains information about the C and C++ programming languages, as supported by IBM, including language extensions for portability and conformance to nonproprietary standards.
IBM XL C/C++ for Linux, V11.1 Optimization and Programming Guide, SC23-8608-00	proguide.pdf	Contains information on advanced programming topics, such as application porting, interlanguage calls with Fortran code, library development, application optimization and parallelization, and the XL C/C++ high-performance libraries.

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

More information related to XL C/C++ including IBM Redbooks® publications,
white papers, tutorials, and other articles, is available on the Web at:

http://www.ibm.com/software/awdtools/xlcpp/linux/library/

For more information about boosting performance, productivity, and portability,
see the C/C++ café at http://www-949.ibm.com/software/rational/cafe/
community/ccpp.

Standards and specifications

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

- *Information Technology - Programming languages - C, ISO/IEC 9899:1990*, also
  known as *C89*.
- *Information Technology - Programming languages - C, ISO/IEC 9899:1999*, also
  known as *C99*.

- *Information Technology - Programming languages - C++, ISO/IEC 14882:1998*, also known as *C++98*.
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2003(E)*, also known as *Standard C++*.
- *Information Technology - Programming languages - Extensions for the programming language C to support new character data types, ISO/IEC DTR 19769*. This draft technical report has been accepted by the C standards committee, and is available at http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1040.pdf.
- *Draft Technical Report on C++ Library Extensions, ISO/IEC DTR 19768*. This draft technical report has been submitted to the C++ standards committee, and is available at http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2005/n1836.pdf.
- *AltiVec Technology Programming Interface Manual*, Motorola Inc. This specification for vector data types, to support vector processing technology, is available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf.
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985*.
- *OpenMP Application Program Interface Version 3.0*, available at http://www.openmp.org

Other IBM information

- *ESSL for AIX V4.4 - ESSL for Linux on POWER V4.4 Guide and Reference*available at the Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL Web page.

Other information

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

Technical support

Additional technical support is available from the XL C/C++ Support page at http://www.ibm.com/software/awdtools/xlcpp/linux/support/. This page provides a portal with search capabilities to a large selection of Technotes and other support information.

If you cannot find what you need, you can send e-mail to compinfo@ca.ibm.com.

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

How to send your comments

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

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

Chapter 1. Compiling and linking applications

By default, when you call the XL C/C++ compiler, all of the following phases of translation are performed:

- preprocessing of program source
- compiling and assembling into object files
- linking into an executable

These different translation phases are actually performed by separate executables, which are referred to as compiler *components*. However, you can use compiler options to perform only certain phases, such as preprocessing, or assembling. You can then reinvoke the compiler to resume processing of the intermediate output to a final executable.

The following sections describe how to invoke the XL C/C++ compiler to preprocess, compile and link source files and libraries:
- "Invoking the compiler"
- "Types of input files" on page 3
- "Types of output files" on page 4
- "Specifying compiler options" on page 5
- "Reusing GNU C/C++ compiler options with **gxlc** and **gxlc++**" on page 10
- "Preprocessing" on page 12
- "Linking" on page 14
- "Compiler messages and listings" on page 16

Invoking the compiler

Different forms of the XL C/C++ compiler invocation commands support various levels of the C and C++ languages. In most cases, you should use the **xlc** command to compile your C source files, and the **xlc++** command to compile C++ source files. Use **xlc++** to link if you have both C and C++ object files.

You can use other forms of the command if your particular environment requires it. Table 4 lists the different basic commands, with the special versions of each basic command. Special commands are described in Table 5 on page 2.

Note: For each invocation command, the compiler configuration file defines default option settings and, in some cases, macros; for information about the defaults implied by a particular invocation, see the /opt/ibmcmp/vac/11.1/etc/ vac.cfg.*$OSRelease.gcc$gccVersion* file for your system. For example, /opt/ibmcmp/vac/11.1/etc/vac.cfg.sles11.gcc432 or /opt/ibmcmp/vac/11.1/etc/ vac.cfg.rhel5.5.gcc412.

Table 4. Compiler invocations

Basic invocations	Description	Equivalent special invocations
xlc	Invokes the compiler for C source files. This command supports all of the ISO C99 standard features, and most IBM language extensions. This invocation is recommended for all applications.	xlc_r

Table 4. Compiler invocations (continued)

Basic invocations	Description	Equivalent special invocations
c99	Invokes the compiler for C source files. This command supports all ISO C99 language features, but does not support IBM language extensions. Use this invocation for strict conformance to the C99 standard.	c99_r
c89	Invokes the compiler for C source files. This command supports all ANSI C89 language features, but does not support IBM language extensions. Use this invocation for strict conformance to the C89 standard.	c89_r
cc	Invokes the compiler for C source files. This command supports pre-ANSI C, and many common language extensions. You can use this command to compile legacy code that does not conform to standard C.	cc_r
gxlc	Invokes the compiler for C source files. This command accepts many common GNU C options, maps them to their XL C option equivalents, and then invokes **xlc**. For more information, refer to "Reusing GNU C/C++ compiler options with **gxlc** and **gxlc++**" on page 10.	
xlc++, xlC	Invokes the compiler for C++ source files. If any of your source files are C++, you must use this invocation to link with the correct runtime libraries. Files with .c suffixes, assuming you have not used the **-+** compiler option, are compiled as C language source code.	xlc++_r, xlC_r
gxlc++, gxlC	Invokes the compiler for C++ files. This command accepts many common GNU C/C++ options, maps them to their XL C/C++ option equivalents, and then invokes **xlc++**. For more information, refer to "Reusing GNU C/C++ compiler options with **gxlc** and **gxlc++**" on page 10.	

Table 5. Suffixes for special invocations

_r-suffixed invocations	All **_r**-suffixed invocations allow for threadsafe compilation and you can use them to link the programs that use multi-threading. Use these commands if you want to create threaded applications.

Command-line syntax

You invoke the compiler using the following syntax, where *invocation* can be replaced with any valid XL C/C++ invocation command listed in Table 4 on page 1:

```
►►──invocation──┬──input_files──────────────────────┬──────────────►◄
                │   └──command_line_options──┘        │
                └─────────────────────────────────────┘
```
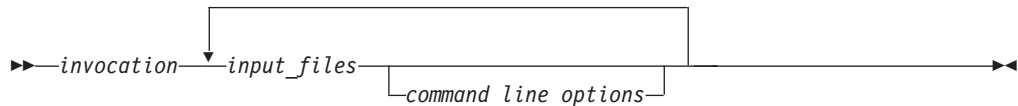
The parameters of the compiler invocation command can be the names of input files, compiler options, and linker options.

Your program can consist of several input files. All of these source files can be compiled at once using only one invocation of the compiler. Although more than one source file can be compiled using a single invocation of the compiler, you can

specify only one set of compiler options on the command line per invocation. Each distinct set of command-line compiler options that you want to specify requires a separate invocation.

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

By default, the invocation command calls *both* the compiler and the linker. It passes linker options to the linker. Consequently, the invocation commands also accept all linker options. To compile without linking, use the **-c** compiler option. The **-c** option stops the compiler after compilation is completed and produces as output, an object file *file_name*.o for each *file_name.nnn* input source file, unless you use the **-o** option to specify a different object file name. The linker is not invoked. You can link the object files later using the same invocation command, specifying the object files without the **-c** option.

Related information
- "Types of input files"

Types of input files

The compiler processes the source files in the order in which they are displayed. If the compiler cannot find a specified source file, it produces an error message and the compiler proceeds to the next specified file. However, the linker does not run and temporary object files are removed.

By default, the compiler preprocesses and compiles all the specified source files. Although you usually want to use this default, you can use the compiler to preprocess the source file without compiling; see "Preprocessing" on page 12 for details.

You can input the following types of files to the XL C/C++ compiler:

C and C++ source files
These are files containing C or C++ source code.

To use the C compiler to compile a C language source file, the source file must have a .c (lowercase c) suffix, unless you compile with the **-qsourcetype=c** option.

To use the C++ compiler, the source file must have a .C (uppercase C), .cc, .cp, .cpp, .cxx, or .c++ suffix, unless you compile with the **-+** or **-qsourcetype=c++** option.

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

Object files
Object files must have a .o suffix, for example, `file_name.`o. Object files, library files, and unstripped executable files serve as input to the linker. After compilation, the linker links all of the specified object files to create an executable file.

Assembler files

Assembler files must have a .s suffix, for example, *file_name*.s, unless you compile with the **-qsourcetype=assembler** option. Assembler files are assembled to create an object file.

Unpreprocessed assembler files

Unpreprocessed assembler files must have a .S suffix, for example, *file_name*.S, unless you compile with the **-qsourcetype=assembler-with-cpp** option. The compiler compiles all source files with a .S extension as if they are assembler language source files that need preprocessing.

Shared library files

Shared library files generally have a .a suffix, for example, *file_name*.a, but they can also have a .so suffix, for example, *file_name*.so.

Unstripped executable files

Executable and linking format (ELF) files that have not been stripped with the operating system **strip** command can be used as input to the compiler.

Related information
- Options summary by functional category: Input control

Types of output files

You can specify the following types of output files when invoking the XL C/C++ compiler:

Executable files

By default, executable files are named a.out. To name the executable file something else, use the **-o** *file_name* option with the invocation command. This option creates an executable file with the name you specify as *file_name*. The name you specify can be a relative or absolute path name for the executable file.

Object files

If you specify the **-c** option, an output object file, *file_name*.o, is produced for each input file. The linker is not invoked, and the object files are placed in your current directory. All processing stops at the completion of the compilation. The compiler gives object files a .o suffix, for example, *file_name*.o, unless you specify the **-o** *file_name* option, giving a different suffix or no suffix at all.

You can link the object files later into a single executable file by invoking the compiler.

Shared library files

If you specify the **-qmkshrobj** option, the compiler generates a single shared library file for all input files. The compiler names the output file a.out, unless you specify the **-o** *file_name* option, and give the file a .so suffix.

Assembler files

If you specify the **-S** option, an assembler file, *file_name*.s, is produced for each input file.

You can then assemble the assembler files into object files and link the object files by reinvoking the compiler.

Preprocessed source files

If you specify the **-P** option, a preprocessed source file, *file_name*.i, is produced for each input file.

You can then compile the preprocessed files into object files and link the object files by reinvoking the compiler.

Listing files

If you specify any of the listing-related options, such as **-qlist** or **-qsource**, a compiler listing file, *file_name*.lst, is produced for each input file. The listing file is placed in your current directory.

Target files

If you specify the **-M** or **-qmakedep** option, a target file suitable for inclusion in a makefile, *file_name*.d is produced for each input file.

Related information
- Options summary by functional category: Output control

Specifying compiler options

Compiler options perform a wide variety of functions, such as setting compiler characteristics, describing the object code and compiler output to be produced, and performing some preprocessor functions. You can specify compiler options in one or more of the following ways:

- On the command line
- In a custom configuration file, which is a file with a .cfg extension
- In your source program
- As system environment variables
- In a makefile

The compiler assumes default settings for most compiler options not explicitly set by you in the ways listed above.

When specifying compiler options, it is possible for option conflicts and incompatibilities to occur. The XL C/C++ compiler resolves most of these conflicts and incompatibilities in a consistent fashion, as follows:

In most cases, the compiler uses the following order in resolving conflicting or incompatible options:

1. Pragma statements in source code override compiler options specified on the command line.
2. Compiler options specified on the command line override compiler options specified as environment variables or in a configuration file. If conflicting or incompatible compiler options are specified in the same command line compiler invocation, the subsequent option in the invocation takes precedence.
3. Compiler options specified as environment variables override compiler options specified in a configuration file.
4. Compiler options specified in a configuration file, command line or source program override compiler default settings.

Option conflicts that do not follow this priority sequence are described in "Resolving conflicting compiler options" on page 8.

Specifying compiler options on the command line

Most options specified on the command line override both the default settings of the option and options set in the configuration file. Similarly, most options specified on the command line are in turn overridden by pragma directives, which

provide you a means of setting compiler options right in the source file. Options that do not follow this scheme are listed in "Resolving conflicting compiler options" on page 8.

There are two kinds of command-line options:
- **-q***option_keyword* (compiler-specific)
- Flag options

-q options

```
►►──-q──option_keyword─────────────────────────────────────────►◄
                       └─=─┬─►──────┬─suboption─┘
                           └────:───┘
```
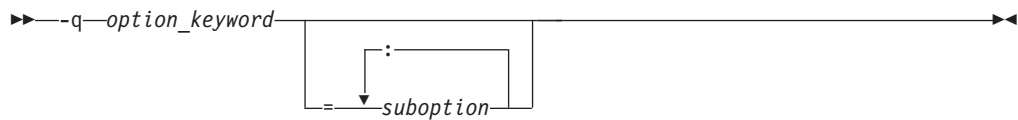
Command-line options in the **-q***option_keyword* format are similar to on and off switches. For *most* **-q** options, if a given option is specified more than once, the last appearance of that option on the command line is the one recognized by the compiler. For example, **-qsource** turns on the source option to produce a compiler listing, and **-qnosource** turns off the source option so no source listing is produced. For example:

```
xlc -qnosource MyFirstProg.c -qsource MyNewProg.c
```

would produce a source listing for both `MyNewProg.c` and `MyFirstProg.c` because the last **source** option specified (**-qsource**) takes precedence.

You can have multiple **-q***option_keyword* instances in the same command line, but they must be separated by blanks. Option keywords can appear in either uppercase or lowercase, but you must specify the **-q** in lowercase. You can specify any **-q***option_keyword* before or after the file name. For example:

```
xlc -qLIST -qfloat=nomaf file.c
xlc file.c -qxref -qsource
```

You can also abbreviate many compiler options. For example, specifying **-qopt** is equivalent to specifying **-qoptimize** on the command line.

Some options have suboptions. You specify these with an equal sign following the **-q***option*. If the option permits more than one suboption, a colon (**:**) must separate each suboption from the next. For example:

```
xlc -qflag=w:e -qattr=full file.c
```

compiles the C source file `file.c` using the option **-qflag** to specify the severity level of messages to be reported. The **-qflag** suboption **w** (warning) sets the minimum level of severity to be reported on the listing, and suboption **e** (error) sets the minimum level of severity to be reported on the terminal. The **-qattr** with suboption **full** will produce an attribute listing of all identifiers in the program.

Flag options
XL C/C++ supports a number of common conventional flag options used on UNIX systems. Lowercase flags are different from their corresponding uppercase flags. For example, **-c** and **-C** are two different compiler options: **-c** specifies that the compiler should only preprocess and compile and not invoke the linker, while **-C** can be used with **-P** or **-E** to specify that user comments should be preserved.

XL C/C++ also supports flags directed to other programming tools and utilities (for example, the **ld** command). The compiler passes on those flags directed to **ld** at link time.

Some flag options have arguments that form part of the flag. For example:

```
xlc stem.c -F/home/tools/test3/new.cfg:xlc
```

where `new.cfg` is a custom configuration file.

You can specify flags that do not take arguments in one string. For example:

```
xlc -Ocv file.c
```

has the same effect as:

```
xlc -O -c -v file.c
```

and compiles the C source file `file.c` with optimization (**-O**) and reports on compiler progress (**-v**), but does not invoke the linker (**-c**).

A flag option that takes arguments can be specified as part of a single string, but you can only use one flag that takes arguments, and it must be the last option specified. For example, you can use the **-o** flag (to specify a name for the executable file) together with other flags, only if the **-o** option and its argument are specified last. For example:

```
xlc -Ovo test test.c
```

has the same effect as:

```
xlc -O -v -otest test.c
```

Most flag options are a single letter, but some are two letters. Note that specifying **-pg** (extended profiling) is not the same as specifying **-p -g** (**-p** for profiling, and **-g** for generating debug information). Take care not to specify two or more options in a single string if there is another option that uses that letter combination.

Specifying compiler options in a configuration file

The default configuration file (/opt/ibmcmp/vac/11.1/etc/vac.cfg.*$OSRelease.gcc$gccVersion*. For example, /opt/ibmcmp/vac/11.1/etc/vac.cfg.sles11.gcc432 or /opt/ibmcmp/vac/11.1/etc/vac.cfg.rhel5.5.gcc412) defines values and compiler options for the compiler. The compiler refers to this file when compiling C or C++ programs. The configuration file is a plain text file. You can edit this file, or create an additional customized configuration file to support specific compilation requirements. For more information, see "Using custom compiler configuration files" on page 32.

Specifying compiler options in program source files

You can specify compiler options within your program source by using pragma directives. A pragma is an implementation-defined instruction to the compiler. For those options that have equivalent pragma directives, you can have several ways to specify the syntax of the pragmas:

- Using **#pragma options** *option_name* syntax — You can use command-line options with the **#pragma options** syntax, which takes the same name as the option, and suboptions with a syntax identical to that of the option. For example, if the command-line option is:

```
-qhalt=w
```

The pragma form is:

```
#pragma options halt=w
```

The descriptions for each individual option indicates whether this form of the pragma is supported. For details, see "#pragma options" on page 356.

- Using **#pragma** *name* syntax — Some options also have corresponding pragma directives that use a pragma-specific syntax, which may include additional or slightly different suboptions. Throughout the section "Individual option descriptions" on page 84, each option description indicates whether this form of the pragma is supported, and the syntax is provided.
- Using the standard C99 `_Pragma` operator — For options that support either forms of the pragma directives listed above, you can also use the C99 `_Pragma` operator syntax in both C and C++.

Complete details on pragma syntax are provided in "Pragma directive syntax" on page 329.

Other pragmas do not have equivalent command-line options; these are described in detail throughout Chapter 5, "Compiler pragmas reference," on page 329.

Options specified with pragma directives in program source files override all other option settings, except other pragma directives. The effect of specifying the same pragma directive more than once varies. See the description for each pragma for specific information.

Pragma settings can carry over into included files. To avoid potential unwanted side effects from pragma settings, you should consider resetting pragma settings at the point in your program source where the pragma-defined behavior is no longer required. Some pragma options offer **reset** or **pop** suboptions to help you do this. These suboptions are listed in the detailed descriptions of the pragmas to which they apply.

Resolving conflicting compiler options

In general, if more than one variation of the same option is specified (with the exception of **-qxref** and **-qattr**), the compiler uses the setting of the last one specified. Compiler options specified on the command line must appear in the order you want the compiler to process them.

Two exceptions to the rules of conflicting options are the **-I***directory* and **-L***directory* options, which have cumulative effects when they are specified more than once.

In most cases, the compiler uses the following order in resolving conflicting or incompatible options:

1. Pragma statements in source code override compiler options specified on the command line.
2. Compiler options specified on the command line override compiler options specified as environment variables or in a configuration file. If conflicting or incompatible compiler options are specified on the command line, the option appearing later on the command line takes precedence.
3. Compiler options specified as environment variables override compiler options specified in a configuration file.
4. Compiler options specified in a configuration file override compiler default settings.

Not all option conflicts are resolved using the preceding rules. The following table summarizes exceptions and how the compiler handles conflicts between them. Rules for resolving conflicts between compiler mode and architecture-specific options are discussed in "Specifying compiler options for architecture-specific, 32-bit or 64-bit compilation."

Option	Conflicting options	Resolution
-qalias=allptrs	-qalias=noansi	-qalias=noansi
-qalias=typeptr	-qalias=noansi	-qalias=noansi
-qhalt	Multiple severities specified by **-qhalt**	Lowest severity specified
-qnoprint	-qxref, -qattr, -qsource, -qlistopt, -qlist	-qnoprint
-qfloat=rsqrt	-qnoignerrno	Last option specified
-qxref	-qxref=full	-qxref=full
-qattr	-qattr=full	-qattr=full
-qfloat=hsflt	-qfloat=spnans	-qfloat=hsflt
-E	-P, -o, -S	-E
-P	-c, -o, -S	-P
-#	-v	-#
-F	-B, -t, -W, -qpath	-B, -t, -W, -qpath
-qpath	-B, -t	-qpath
-S	-c	-S
-qnostdinc	-qc_stdinc, -qcpp_stdinc, -qgcc_c_stdinc, -qgcc_cpp_stdinc	-qnostdinc

Specifying compiler options for architecture-specific, 32-bit or 64-bit compilation

You can use the **-q32**, **-q64**, **-qarch**, and **-qtune** compiler options to optimize the output of the compiler to suit:
*   The broadest possible selection of target processors
*   A range of processors within a given processor architecture family
*   A single specific processor

Generally speaking, the options do the following:

*   **-q32** selects 32-bit execution mode.

*   **-q64** selects 64-bit execution mode.

*   **-qarch** selects the general family processor architecture for which instruction code should be generated. Certain **-qarch** settings produce code that will run *only* on systems that support *all* of the instructions generated by the compiler in response to a chosen **-qarch** setting.

*   **-qtune** selects the specific processor for which compiler output is optimized. Some **-qtune** settings can also be specified as **-qarch** options, in which case they do not also need to be specified as a **-qtune** option. The **-qtune** option influences only the performance of the code when running on a particular system but does not determine where the code will run.

The compiler evaluates compiler options in the following order, with the last allowable one found determining the compiler mode:

1.   Internal default (32-bit mode)

2. Configuration file settings

3. Command line compiler options (**-q32**, **-q64**, **-qarch**, **-qtune**)

4. Source file statements (**#pragma options tune=***suboption*)

The compilation mode actually used by the compiler depends on a combination of the settings of the **-q32**, **-q64**, **-qarch** and **-qtune** compiler options, subject to the following conditions:

- *Compiler mode* is set according to the last-found instance of the **-q32** or **-q64** compiler options.

- *Architecture target* is set according to the last-found instance of the **-qarch** compiler option, provided that the specified **-qarch** setting is compatible with the *compiler mode* setting. If the **-qarch** option is not set, the compiler sets **-qarch** to the appropriate default based on the effective compiler mode setting. See "-qarch" on page 95 for details.

- Tuning of the architecture target is set according to the last-found instance of the **-qtune** compiler option, provided that the **-qtune** setting is compatible with the *architecture target* and *compiler mode* settings. If the **-qtune** option is not set, the compiler assumes a default **-qtune** setting according to the **-qarch** setting in use. If **-qarch** is not specified, the compiler sets **-qtune** to the appropriate default based on the effective **-qarch** as selected by default based on the effective compiler mode setting.

Allowable combinations of these options are found in "-qtune" on page 308.

The following list describes possible option conflicts and compiler resolution of these conflicts:

- **-q32** or **-q64** setting is incompatible with user-selected **-qarch** option.

  Resolution: -q32 or **-q64** setting overrides **-qarch** option; compiler issues a warning message, sets **-qarch** to its default setting, and sets the **-qtune** option accordingly to its default value.

- **-qarch** option is incompatible with user-selected **-qtune** option.

  Resolution: Compiler issues a warning message, and sets **-qtune** to the **-qarch** setting's default **-qtune** value.

- Selected **-qarch** or **-qtune** options are not known to the compiler.

  Resolution: Compiler issues a warning message, sets **-qarch** and **-qtune** to their default settings. The compiler mode (32-bit or 64-bit) is determined by the **-q32**/**-q64** compiler settings.

Related information
- "-qarch" on page 95
- "-qtune" on page 308
- "-q32, -q64" on page 87

Reusing GNU C/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 use to invoke the compiler. These utilities are provided to facilitate the reuse of make files created for applications previously developed with GNU C/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 /opt/ibmcmp/vac/11.1/etc/gxlc.cfg. The GNU C/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 **gxlc** or **gxlc++** option mapping" on page 36.

gxlc or gxlc++ syntax

The following diagram shows the **gxlc** or **gxlc++** syntax:

where:

filename
        Is the name of the file to be compiled.

-v   Verifies the command that is used to invoke XL C/C++. The utility displays the XL C/C++ invocation command that it has created, before using it to invoke the compiler.

-vv   Runs a simulation. The utility 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. The utility 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* are delimited by a comma.

-*gcc_or_g++_options*
        The GNU C/C++ options that are translated to XL C/C++ options. The utility emits a warning for any option it cannot translate. The GNU C/C++ options that are currently recognized by **gxlc** or **gxlc++** are in the configuration file gxlc.cfg. Multiple *-gcc_or_g++_options* are delimited by the space character.

Example

To use the GCC **-fstrict-aliasing** option to compile the C version of the Hello World program, you can use:

```
gxlc -fstrict-aliasing hello.c
```

which translates into:

```
xlc -qalias=ansi hello.c
```

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

Related information
• "Configuring the **gxlc** or **gxlc++** option mapping" on page 36

Preprocessing

Preprocessing manipulates the text of a source file, usually as a first phase of translation that is initiated by a compiler invocation. Common tasks accomplished by preprocessing are macro substitution, testing for conditional compilation directives, and file inclusion.

You can invoke the preprocessor separately to process text without compiling. The output is an intermediate file, which can be input for subsequent translation. Preprocessing without compilation can be useful as a debugging aid because it provides a way to see the result of include directives, conditional compilation directives, and complex macro expansions.

The following table lists the options that direct the operation of the preprocessor.

Option	Description
"-E" on page 127	Preprocesses the source files and writes the output to standard output. By default, #line directives are generated.
"-P" on page 232	Preprocesses the source files and creates an intermediary file with a .i file name suffix for each source file. By default, #line directives are not generated.
"-qppline" on page 244	Toggles on and off the generation of #line directives for the **-E** and **-P** options.
"-C, -C!" on page 106	Preserves comments in preprocessed output.
"-D" on page 120	Defines a macro name from the command line, as if in a #define directive.
"-U" on page 311	Undefines a macro name defined by the compiler or by the **-D** option.
"-qshowmacros" on page 267	Emits macro definitions to preprocessed output.

Directory search sequence for include files

The XL C/C++ compiler supports the following types of include files:
- Header files supplied by the compiler (referred to throughout this document as *XL C/C++ headers*)
- Header files mandated by the C and C++ standards (referred to throughout this document as *system headers*)
- Header files supplied by the operating system (also referred to throughout this document as *system headers*)
- User-defined header files

You can use any of the following methods to include any type of header file:
- Use the standard #include <*file_name*> preprocessor directive in the including source file.
- Use the standard #include "*file_name*" preprocessor directive in the including source file.
- Use the **-qinclude** compiler option.

If you specify the header file using a full (absolute) path name, you can use these methods interchangeably, regardless of the type of header file you want to include.

However, if you specify the header file using a *relative* path name, the compiler uses a different directory search order for locating the file depending on the method used to include the file.

Furthermore, the **-qidirfirst** and **-qstdinc** compiler options can affect this search order. The following summarizes the search order used by the compiler to locate header files depending on the mechanism used to include the files and on the compiler options that are in effect:

1. Header files included with **-qinclude** only: The compiler searches the current (working) directory from which the compiler is invoked.[1]
2. Header files included with **-qinclude** or `#include "file_name"`: The compiler searches the directory in which the including file is located.[1]
3. All header files: The compiler searches each directory specified by the **-I** compiler option, in the order that it displays on the command line.
4. All header files: ► C The compiler searches the standard directory for the XL C headers. The default directory for these headers is specified in the compiler configuration file. This is normally /opt/ibmcmp/vac/11.1/include/, but the search path can be changed with the **-qc_stdinc** compiler option. ► C++ The compiler searches the standard directory for the XL C++ headers. The default directory for these headers is specified in the compiler configuration file. This is normally /opt/ibmcmp/vacpp/11.1/include/, but the search path can be changed with the **-qcpp_stdinc** compiler option.[2]
5. All header files: ► C The compiler searches the standard directory for the system headers. The default directory for these headers is specified in the compiler configuration file. The default location for these headers is specified in the compiler configuration file. This location is set during installation, but the search path can be changed with the **-qgcc_c_stdinc** option. ► C++ The compiler searches the standard directory for the system headers. The default directory for these headers is specified in the compiler configuration file. The default location for these headers is specified in the compiler configuration file. This location is set during installation, but the search path can be changed with the **-qgcc_cpp_stdinc** option.[2]

Note:

1. If the **-qidirfirst** compiler option is in effect, step 3 is performed before steps 1 and 2.
2. If the **-qnostdinc** compiler option is in effect, steps 4 and 5 are omitted.

Related information
- "-I" on page 158
- "-qc_stdinc (C only)" on page 118
- "-qcpp_stdinc (C++ only)" on page 119
- "-qgcc_c_stdinc (C only)" on page 150
- "-qgcc_cpp_stdinc (C++ only)" on page 151
- "-qidirfirst" on page 159
- "-qinclude" on page 162
- "-qstdinc" on page 285

Linking

The linker links specified object files to create one executable file. Invoking the compiler with one of the invocation commands automatically calls the linker unless you specify one of the following compiler options: **-E**, **-P**, **-c**, **-S**, **-qsyntaxonly** or **-#**.

Input files
> Object files, unstripped executable files, and library files serve as input to the linker. Object files must have a .o suffix, for example, `filename.o`. Static library file names have an .a suffix, for example, `filename.a`. Dynamic library file names typically have a .so suffix, for example, `filename.so`.

Output files
> The linker generates an *executable file* and places it in your current directory. The default name for an executable file is a.out. To name the executable file explicitly, use the **-o** file_name option with the compiler invocation command, where *file_name* is the name you want to give to the executable file. For example, to compile `myfile.c` and generate an executable file called `myfile`, enter:
>
> ```
> xlc myfile.c -o myfile
> ```
>
> If you use the **-qmkshrobj** option to create a shared library, the default name of the shared object created is a.out. You can use the **-o** option to rename the file and give it a .so suffix.

You can invoke the linker explicitly with the **ld** command. However, the compiler invocation commands set several linker options, and link some standard files into the executable output by default. In most cases, it is better to use one of the compiler invocation commands to link your object files. For a complete list of options available for linking, see "Linking" on page 81.

Related information

"-qmkshrobj" on page 224

Order of linking

The compiler links libraries in the following order:
1. System startup libraries
2. User .o files and libraries
3. XL C/C++ libraries
4. C++ standard libraries
5. C standard libraries

Related information
- "Linking" on page 81
- "Redistributable libraries"

Redistributable libraries

If you build your application using XL C/C++, it might use one or more of the following redistributable libraries. If you ship the application, ensure that the users of the application have the packages containing the libraries. To make sure the required libraries are available to users, you must do one of the following:

- You can ship the packages that contain the redistributable libraries with the application. The packages are stored under the images/rpms directory on the installation CD.
- The user can download the packages that contain the redistributable libraries from the XL C/C++ support Web site at:

  http://www.ibm.com/software/awdtools/xlcpp/linux/support/

For information about the licensing requirements related to the distribution of these packages see the LicAgree.pdf file on the CD.

Table 6. Redistributable libraries

Package name	Libraries (and default installation path)	Description
vac.lib	/opt/ibmcmp/vac/V11.1/lib/libxl.a /opt/ibmcmp/vac/V11.1/lib64/libxl.a /opt/ibmcmp/vac/V11.1/lib/libxlopt.a /opt/ibmcmp/vac/V11.1/lib64/libxlopt.a	XL C compiler libraries
vacpp.rte	/opt/ibmcmp/vac/V11.1/lib/libibmc++.so.1 /opt/ibmcmp/vac/V11.1/lib64/libibmc++.so.1	XL C++ runtime libraries
xlsmp.rte	/opt/ibmcmp/lib/libxlomp_ser.so.1 /opt/ibmcmp/lib/libxlsmp.so.1 /opt/ibmcmp/lib64/libxlomp_ser.so.1 /opt/ibmcmp/lib64/libxlsmp.so.1	SMP (OMP) runtime libraries
xlsmp.msg.rte	/opt/ibmcmp/msg/en_US/smprt.cat /opt/ibmcmp/msg/en_US.utf8/smprt.cat	SMP message catalogs (English)
	/opt/ibmcmp/msg/ja_JP/smprt.cat /opt/ibmcmp/msg/ja_JP.eucjp/smprt.cat /opt/ibmcmp/msg/ja_JP.utf8/smprt.cat	SMP message catalogs (Japanese)
	/opt/ibmcmp/msg/zh_CN/smprt.cat /opt/ibmcmp/msg/zh_CN.gb18030/smprt.cat /opt/ibmcmp/msg/zh_CN.gb2312/smprt.cat /opt/ibmcmp/msg/zh_CN.gbk/smprt.cat /opt/ibmcmp/msg/zh_CN.utf8/smprt.cat	SMP message catalogs (Chinese)

Backward compatibility issues

This section describes backward compatibility issues and their workarounds.

Compiler option compatibility issues

In IBM XL C/C++ for Linux, V11.1, the implementation of the threadprivate data, that is, OpenMP threadprivate variable, has been improved. The operating system thread local storage is used instead of the runtime implementation. The new implementation might improve performance on some applications.

If you plan to mix the object files **.o** that you have compiled with levels prior to 11.1 with the object files that you compiled with IBM XL C/C++ for Linux, V11.1, and the same OpenMP threadprivate variables are referenced in both old and new object files, different implementations might cause incompatibility issues. A link error, a compile time error or other undefined behaviors might occur. To support backward compatibility, you can use the **-qsmp=noostls** suboption to switch back to the old implementation. You can recompile the entire program with the default suboption **-qsmp=ostls** to get the benefit of the new implementation.

If you are not sure whether the object files you have compiled with levels prior to IBM XL C/C++ for Linux, V11.1 contain any old implementation, you can use the **readelf -s** command to determine whether you need to use the **-qsmp=noostls** suboption. The following code is an example that shows how to use the **readelf -s** command:

```
> readelf -s oldfiles.o
...
._xlGetThStorageBlock U          -
._xlGetThValue        U          -
...
```

In the preceding example, if _xlGetThStorageBlock or _xlGetThValue is found, this means the object files contain old implementation. In this case, you must use **-qsmp=noostls**; otherwise, use the default suboption **-qsmp=ostls**.

Compiler messages and listings

The following sections discuss the various methods of reporting provided by the compiler after compilation.

- "Compiler messages"
- "Compiler return codes" on page 18
- "Compiler listings" on page 19
- "Message catalog errors" on page 21
- "Paging space errors during compilation" on page 22

Compiler messages

When the compiler encounters a programming error while compiling a C or C++ source program, it issues a diagnostic message to the standard error device, or to a listing file if you compile with the **-qsource** option. These diagnostic messages are specific to the C or C++ language.

▶ C If you specify the compiler option **-qsrcmsg** and the error is applicable to a particular line of code, the reconstructed source line or partial source line is included with the error message. A reconstructed source line is a preprocessed source line that has all the macros expanded.

You can control the diagnostic messages issued, according to their severity, using either the **-qflag** option or the **-w** option. To get additional informational messages about potential problems in your program, use the **-qinfo** option.

Related information
- "-qsource" on page 276
- "-qsrcmsg (C only)" on page 280
- "-qflag" on page 136
- "-w" on page 320
- "-qinfo" on page 164

Compiler message format

Diagnostic messages have the following format:

```
"file", line line_number.column_number: 15dd-number (severity) text.
```

where:

file Is the name of the C or C++ source file with the error.

line_number
> Is the source code line number where the error was found.

column_number
> Is the source code column number where the error was found.

15 Is the compiler product identifier.

dd Is a two-digit code indicating the compiler component that issued the message. *dd* can have the following values:

> **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)

number
> Is the message number.

severity
> Is a letter representing the severity of the error. See "Message severity levels and compiler response" for a description of these.

text
> Is a message describing the error.

▶ `C`   If you compile with **-qsrcmsg**, diagnostic messages have the following format:

`x - 15dd-nnn(severity) text.`

where *x* is a letter referring to a finger in the finger line.

Message severity levels and compiler response

The XL C/C++ compiler uses a multilevel classification scheme for diagnostic messages. Each level of severity is associated with a compiler response. The following table provides a key to the abbreviations for the severity levels and the associated default compiler response. You can adjust the default compiler response by using any of the following options:

- **-qhalt** halts the compilation phase at a lower severity level than the default
- **-qmaxerr** halts the compilation phase as soon as a specific number of errors at a specific severity level is reached
- ▶ `C++`   **-qhaltonmsg** halts the compilation phase as soon as a specific error is encountered

Table 7. Compiler message severity levels

Letter	Severity	Compiler response
I	Informational	Compilation continues and object code is generated. The message reports conditions found during compilation.
W	Warning	Compilation continues and object code is generated. The message reports valid but 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.

Table 7. Compiler message severity levels (continued)

Letter	Severity	Compiler response
S	Severe error	Compilation continues, but object code is not generated. Error conditions exist that the compiler cannot correct: • If the message indicates a resource limit (for example, file system full or paging space full), provide additional resources and recompile. • If the message indicates that different compiler options are needed, recompile using them. • Check for and correct any other errors reported prior to the severe error. • If the message indicates an internal compiler error, the message should be reported to your IBM service representative.
▶ C U	Unrecoverable error	The compiler halts. An internal compiler error has occurred. The message should be reported to your IBM service representative.

Related information
- "-qhalt" on page 153
- "-qmaxerr" on page 219
- "-qhaltonmsg (C++ only)" on page 154
- Options summary by functional category: Listings and messages

Compiler 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.
- ▶ C++ No message specified by the **-qhaltonmsg** compiler option is issued.

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

Return code	Error type
1	Any error with a severity level higher than the setting of the **-qhalt** compiler option has been detected.
40	An option error or an unrecoverable error has been detected.
41	A configuration file error has been detected.
249	A no-files-specified error has been detected.
250	An out-of-memory error has been detected. The compiler cannot allocate any more memory for its use.
251	A signal-received error has been detected. That is, an unrecoverable error or interrupt signal has occurred.
252	A file-not-found error has been detected.
253	An input/output error has been detected: files cannot be read or written to.
254	A fork error has been detected. A new process cannot be created.

255	An error has been detected while the process was running.

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

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 **gxlc** or **gxlc++** option error or unrecoverable error has been detected.
255   An error has been detected while the process was running.

Compiler listings

A listing is a compiler output file (with a .lst suffix) that contains information about a particular compilation. As a debugging aid, a compiler listing is useful for determining what has gone wrong in a compilation. For example, any diagnostic messages emitted during compilation are written to the listing.

To produce a listing, you can compile with any of the following options, which provide different types of information:

- -qsource
- -qlistopt
- -qattr
- -qxref
- -qlist
- -qreport

When any of these options is in effect, a listing file *filename*.lst is saved in the current directory for every input file named in the compilation.

Listing information is organized in sections. A listing contains a header section and a combination of other sections, depending on other options in effect. The contents of these sections are described as follows.

Header section
> Lists the compiler name, version, release, the source file name, and the date and time of the compilation.

Source section
> If you use the **-qsource** option, lists the input source code with line numbers. If there is an error at a line, the associated error message appears after the source line. Lines containing macros have additional lines showing the macro expansion. By default, this section only lists the main source file. Use the **-qshowinc** option to expand all header files as well.

Options section
> Lists the non-default options that were in effect during the compilation. To list all options in effect, specify the **-qlistopt** option.

Attribute and cross-reference listing section
> If you use the **-qattr** or **-qxref** options, provides information about the variables used in the compilation unit, such as type, storage duration, scope, and where they are defined and referenced. Each of these options provides different information on the identifiers used in the compilation.

File table section

Lists the file name and number for each main source file and include file. Each file is associated with a file number, starting with the main source file, which is assigned file number 0. For each file, the listing shows from which file and line the file was included. If the **-qshowinc** option is also in effect, each source line in the source section will have a file number to indicate which file the line came from.

PDF report section

The following information is included in this section when you use the **-qreport** option with the **-qpdf2** option:

Loop iteration count

The most frequent loop iteration count and the average iteration count, for a given set of input data, are calculated for most loops in a program. This information is only available when the program is compiled at optimization level **-O5**.

Block and call count

This section of the report covers the *Call Structure* of the program and the respective execution count for each called function. It also includes *Block information* for each function. For non-user defined functions, only execution count is given. The Total Block and Call Coverage, and a list of the user functions ordered by decreasing execution count are printed in the end of this report section. In addition, the Block count information is printed at the beginning of each block of the pseudo-code in the listing files.

Cache miss

This section of the report is printed in a single table. It reports the number of *Cache Misses* for certain functions, with additional information about the functions such as: `Cache Level`, `Cache Miss Ratio`, `Line Number`, `File Name`, and `Memory Reference`.

Note: You must use the option **-qpdf1=level=2** to get this report. You can also select the level of cache to profile using the environment variable **PDF_PM_EVENT** during run time.

Transformation report section

If the **-qreport** option is in effect, this section displays pseudo code that corresponds to the original source code, so that you can see parallelization and loop transformations that the **-qhot** or **-qsmp** option has generated. This section of the report will also show additional loop transformation and parallelization information on loop nests if you compile with **-qsmp** and **-qhot=level=2**.

This section also reports the number of streams created for a given loop and the location of data prefetch instructions inserted by the compiler. To generate information about data prefetch insertion locations, use the optimization level of **-qhot**, **-O3 -qhot**, **-O4** or **-O5** together with **-qreport**.

Data reorganization section

Displays data reorganization messages for program variable data during the IPA link pass when **-qreport** is used with **-qipa=level=2** or **-O5**. Reorganization information includes:
- array splitting
- array transposing
- memory allocation merging

- array interleaving
- array coalescing

Compilation epilogue section

Displays a summary of the diagnostic messages by severity level, the number of source lines read, and whether or not the compilation was successful.

Object section

If you use the **-qlist** option, lists the object code generated by the compiler. This section is useful for diagnosing execution time problems, if you suspect the program is not performing as expected due to code generation error.

Related information
- Summary of command line options: Listings and messages

Message catalog errors

Before the compiler can compile your program, the message catalogs must be installed and the environment variables *LANG* and *NLSPATH* must be set to a language for which the message catalog has been installed.

If you see the following message during compilation, the appropriate message catalog cannot be opened:

```
Error occurred while initializing the message system in
file: message_file
```

where *message_file* is the name of the message catalog that the compiler cannot open. This message is issued in English only.

You must then verify that the message catalogs and the environment variables are in place and correct. If the message catalog or environment variables are not correct, compilation can continue, but diagnostic messages are suppressed and the following message is issued instead:

```
No message text for message_number
```

where *message_number* is the compiler internal message number. This message is issued in English only.

To determine which message catalogs are installed on your system, assuming that you have installed the compiler to the default location, you can list all of the file names for the catalogs by the following command:

```
ls /opt/ibmcmp/vacpp/11.1/msg/$LANG/*.cat
```

where *LANG* is the environment variable on your system that specifies the system locale.

The compiler calls the message catalogs for **en_US** by default if *LANG* is not set correctly.

For more information about the *NLSPATH* and *LANG* environment variables, see your operating system documentation.

Paging space errors during compilation

If the operating system runs low on paging space during a compilation, the compiler issues the following message:

```
1501-229 Compilation ended due to lack of space.
```

To minimize paging-space problems, do any of the following and recompile your program:

- Reduce the size of your program by splitting it into two or more source files
- Compile your program without optimization
- Reduce the number of processes competing for system paging space
- Increase the system paging space

For more information about paging space and how to allocate it, see your operating system documentation.

Chapter 2. Configuring compiler defaults

When you compile an application with XL C/C++, the compiler uses default settings that are determined in a number of ways:

- Internally defined settings. These settings are predefined by the compiler and you cannot change them.
- Settings defined by system environment variables. Certain environment variables are required by the compiler; others are optional. You might have already set some of the basic environment variables during the installation process (for more information, see the XL C/C++ Installation Guide). "Setting environment variables" provides a complete list of the required and optional environment variables you can set or reset after installing the compiler, including those used for parallel processing.
- Settings defined in the compiler configuration file, vac.cfg. The compiler requires many settings that are determined by its configuration file. Normally, the configuration file is automatically generated during the installation procedure. (For more information, see the XL C/C++ Installation Guide). However, you can customize this file after installation, to specify additional compiler options, default option settings, library search paths, and other settings. Information on customizing the configuration file is provided in "Using custom compiler configuration files" on page 32.
- Settings defined by the GCC options configuration file. If you are using the **gxlc** or **gxlc++** utility to map GCC options, the default option mappings are defined in the /opt/ibmcmp/vac/11.1/etc/gxlc.cfg file. You can customize this file to suit your requirements; for more information, see "Configuring the **gxlc** or **gxlc++** option mapping" on page 36.

Setting environment variables

To set environment variables in Bourne, Korn, and BASH shells, use the following commands:

```
variable=value
export variable
```

where *variable* is the name of the environment variable, and *value* is the value you assign to the variable.

To set environment variables in the C shell, use the following command:

```
setenv variable value
```

where *variable* is the name of the environment variable, and *value* is the value you assign to the variable.

To set the variables so that all users have access to them, in Bourne, Korn, and BASH shells, add the commands to the file */etc/profile*. To set them for a specific user only, add the commands to the file .profile in the user's home directory. In C shell, add the commands to the file */etc/csh.cshrc*. To set them for a specific user only, add the commands to the file .cshrc in the user's home directory. The environment variables are set each time the user logs in.

The following sections discuss the environment variables you can set for XL C/C++ and applications you have compiled with it:

- "Compile-time and link-time environment variables"
- "Runtime environment variables"

Compile-time and link-time environment variables

The following environment variables are used by the compiler when you are compiling and linking your code. Many are built into the Linux operating system. With the exception of *LANG* and *NLSPATH*, which must be set if you are using a locale other than the default en_US, all of these variables are optional.

LANG

Specifies the locale for your operating system. The default locale used by the compiler for messages and help files is United States English, en_US, but the compiler supports other locales. For a list of these, see *National language support* in the *XL C/C++ Installation Guide*. For more information on setting the *LANG* environment variable to use an alternate locale, see your operating system documentation.

LD_RUN_PATH

Specifies search paths for dynamically loaded libraries, equivalent to using the **-R** link-time option. The shared-library locations named by the environment variable are embedded into the executable, so the dynamic linker can locate the libraries at application run time. For more information about this environment variable, see your operating system documentation. See also "-R" on page 253.

NLSPATH

Specifies the directory search path for finding the compiler message and help files. You only need to set this environment variable if the national language to be used for the compiler message and help files is not English. For information on setting the *NLSPATH*, see *Enabling the XL C/C++ error messages* in the *XL C/C++ Installation Guide*.

PATH Specifies the directory search path for the executable files of the compiler. Executables are in /opt/ibmcmp/vacpp/11.1/bin/ if installed to the default location. For information, see *Setting the PATH environment variable to include the path to the XL C/C++ invocations* in the *XL C/C++ Installation Guide*

TMPDIR

Optionally specifies the directory in which temporary files are created during compilation. The default location, */tmp/*, may be inadequate at high levels of optimization, where paging and temporary files can require significant amounts of disk space, so you can use this environment variable to specify an alternate directory.

XLC_USR_CONFIG

Specifies the location of a custom configuration file to be used by the compiler. The file name must be given with its absolute path. The compiler will first process the definitions in this file before processing those in the default system configuration file, or those in a customized file specified by the **-F** option; for more information, see "Using custom compiler configuration files" on page 32.

Runtime environment variables

The following environment variables are used by the system loader or by your application when it is executed. All of these variables are optional.

LD_LIBRARY_PATH

Specifies an alternate directory search path for dynamically linked libraries at application run time. If shared libraries required by your application have been moved to an alternate directory that was not specified at link time, and you do not want to relink the executable, you can set this environment variable to allow the dynamic linker to locate them at run time. For more information about this environment variable, see your operating system documentation.

PDFDIR

Optionally specifies the directory in which profiling information is saved when you run an application that you have compiled with the **-qpdf1** option. The default value is unset, and the compiler places the profile data file in the current working directory. When you recompile or relink your application with **-qpdf2**, the compiler uses the data saved in this directory to optimize the application. It is recommended that you set this variable to an absolute path if you will be using profile-directed feedback. See "-qpdf1, -qpdf2" on page 236 for more information.

PDF_BIND_PROCESSOR

If you want to bind your process to a particular processor, you can specify *PDF_BIND_PROCESSOR* to bind the process tree from the executable to a different processor. Processor 0 is set by default.

Environment variables for parallel processing

The XLSMPOPTS environment variable sets options for program run time using loop parallelization. Suboptions for the XLSMPOPTS environment variables are discussed in detail in "XLSMPOPTS."

If you are using OpenMP constucts for parallelization, you can also specify runtime options using OMP environment variables, as discussed in "OpenMP environment variables for parallel processing" on page 29.

When runtime options specified by OMP and XLSMPOPTS environment variables conflict, OMP options will prevail.

Note: You must use a threadsafe compiler mode invocations when compiling parallelized program code.

Related information
* "Pragma directives for parallel processing" on page 377
* "Built-in functions for parallel processing" on page 506

XLSMPOPTS

Runtime options affecting parallel processing can be specified with the XLSMPOPTS environment variable. This environment variable must be set before you run an application, and uses basic syntax of the form:

```
                                        ┌──:────────────────────┐
►►──XLSMPOPTS── =──┬────┬──▼──runtime_option_name── =────option_setting──┴──┬────┬──►◄
                   └─"─┘                                                     └─"─┘
```

You can specify option names and settings in uppercase or lowercase. You can add blanks before and after the colons and equal signs to improve readability.

However, if the XLSMPOPTS option string contains imbedded blanks, you must enclose the entire option string in double quotation marks (").

For example, to have a program run time create 4 threads and use dynamic scheduling with chunk size of 5, you would set the XLSMPOPTS environment variable as shown below:

```
XLSMPOPTS=PARTHDS=4:SCHEDULE=DYNAMIC=5
```

The following are the available runtime option settings for the XLSMPOPTS environment variable:

Scheduling options are as follows:

schedule

Specifies the type of scheduling algorithms and chunk size (n) that are used for loops to which no other scheduling algorithm has been explicitly assigned in the source code.

Work is assigned to threads in a different manner, depending on the scheduling type and chunk size used. Choosing chunking granularity is a tradeoff between overhead and load balancing. The syntax for this option is **schedule**=*suboption*, where the suboptions are defined as follows:

affinity[=n]

The iterations of a loop are initially divided into n partitions, containing **ceiling**(*number_of_iterations/number_of_threads*) iterations. Each partition is initially assigned to a thread and is then further subdivided into chunks that each contain n iterations. If n is not specified, then the chunks consist of **ceiling**(*number_of_iterations_left_in_partition* / 2) loop iterations.

When a thread becomes free, it takes the next chunk from its initially assigned partition. If there are no more chunks in that partition, then the thread takes the next available chunk from a partition initially assigned to another thread.

The work in a partition initially assigned to a sleeping thread will be completed by threads that are active.

The **affinity** scheduling type does not appear in the OpenMP API standard.

dynamic[=n]

The iterations of a loop are divided into chunks containing n iterations each. If n is not specified, then the chunks consist of **ceiling**(*number_of_iterations/number_of_threads*) iterations.

Active threads are assigned these chunks on a "first-come, first-do" basis. Chunks of the remaining work are assigned to available threads until all work has been assigned.

If a thread is asleep, its assigned work will be taken over by an active thread once that thread becomes available.

guided[=n]

The iterations of a loop are divided into progressively smaller chunks until a minimum chunk size of n loop iterations is reached. If n is not specified, the default value for n is 1 iteration.

Active threads are assigned chunks on a "first-come, first-do" basis. The first chunk contains **ceiling**(*number_of_iterations*/*number_of_threads*) iterations. Subsequent chunks consist of **ceiling**(*number_of_iterations_left* / *number_of_threads*) iterations.

static[=*n*]
    The iterations of a loop are divided into chunks containing *n* iterations each. Each thread is assigned chunks in a "round-robin" fashion. This is known as *block cyclic scheduling*. If the value of *n* is 1, then the scheduling type is specifically referred to as *cyclic scheduling*.

    If *n* is not specified, the chunks will contain **ceiling**(*number_of_iterations*/*number_of_threads*) iterations. Each thread is assigned one of these chunks. This is known as *block scheduling*.

    If a thread is asleep and it has been assigned work, it will be awakened so that it may complete its work.

n   Must be an integral assignment expression of value 1 or greater.

Specifying **schedule** with no suboption is equivalent to **schedule=runtime**.

Parallel environment options are as follows:

parthds=*num*
    Specifies the number of threads (*num*) requested, which is usually equivalent to the number of processors available on the system.

    Some applications cannot use more threads than the maximum number of processors available. Other applications can experience significant performance improvements if they use more threads than there are processors. This option gives you full control over the number of user threads used to run your program.

    The default value for *num* is the number of processors available on the system.

usrthds=*num*
    Specifies the maximum number of threads (*num*) that you expect your code will explicitly create if the code does explicit thread creation. The default value for *num* is 0.

stack=*num*
    Specifies the largest amount of space in bytes (*num*) that a thread's stack needs. The default value for *num* is 2097152.

    Set *num* so it is within the acceptable upper limit. *num* can be up to 256 MB for 32-bit mode, or up to the limit imposed by system resources for 64-bit mode. An application that exceeds the upper limit may cause a segmentation fault.

    The glibc library is compiled by default to allow a stack size of 2 MB. Setting num to a value greater than this will cause the default stack size to be used. If larger stack sizes are required, you should link the program to a glibc library compiled with the FLOATING_STACKS parameter turned on.

stackcheck[=*num*]
    When the **-qsmp=stackcheck** is in effect, enables stack overflow checking for slave threads at runtime. *num* is the size of the stack in bytes; when the remaining stack size is less than this value, a runtime warning message is issued. If you do not specify a value for *num*, the default value is 4096

bytes. Note that this option only has an effect when the **-qsmp=stackcheck** has also been specified at compile time. See "-qsmp" on page 272 for more information.

startproc=*cpu_id*
Enables thread binding and specifies the *cpu_id* to which the first thread binds. If the value provided is outside the range of available processors, a warning message is issued and no threads are bound.

procs=cpu_id*[,cpu_id,...]*
Enables thread binding and specifies a list of *cpu_id* to which the threads are bound. If the number of CPU IDs specified is less than the number of threads used by the program, the remaining threads are not bound.

stride=*num*
Specifies the increment used to determine the *cpu_id* to which subsequent threads bind. *num* must be greater than or equal to 1. If the value provided causes a thread to bind to a CPU outside the range of available processors, a warning message is issued and no threads are bound.

Performance tuning options are as follows:

spins=*num*
Specifies the number of loop spins, or iterations, before a yield occurs.

When a thread completes its work, the thread continues executing in a tight loop looking for new work. One complete scan of the work queue is done during each busy-wait state. An extended busy-wait state can make a particular application highly responsive, but can also harm the overall responsiveness of the system unless the thread is given instructions to periodically scan for and yield to requests from other applications.

A complete busy-wait state for benchmarking purposes can be forced by setting both **spins** and **yields** to 0.

The default value for *num* is 100.

yields=*num*
Specifies the number of yields before a sleep occurs.

When a thread sleeps, it completely suspends execution until another thread signals that there is work to do. This provides better system utilization, but also adds extra system overhead for the application.

The default value for *num* is 100.

delays=*num*
Specifies a period of do-nothing delay time between each scan of the work queue. Each unit of delay is achieved by running a single no-memory-access delay loop.

The default value for *num* is 500.

Dynamic profiling options are as follows:

profilefreq=*n*
Specifies the frequency with which a loop should be revisited by the dynamic profiler to determine its appropriateness for parallel or serial execution.The runtime library uses dynamic profiling to dynamically tune the performance of automatically parallelized loops. Dynamic profiling gathers information about loop running times to determine if the loop should be run sequentially or in

parallel the next time through. Threshold running times are set by the **parthreshold** and **seqthreshold** dynamic profiling options, described below.

The allowed values for this option are the numbers from 0 to 32. If *num* is 0, all profiling is turned off, and overheads that occur because of profiling will not occur. If *num* is greater than 0, running time of the loop is monitored once every *num* times through the loop. The default for *num* is 16. Values of *num* exceeding 32 are changed to 32.

It is important to note that dynamic profiling is not applicable to user-specified parallel loops.

parthreshold=_num_
Specifies the time, in milliseconds, below which each loop must execute serially. If you set *num* to 0, every loop that has been parallelized by the compiler will execute in parallel. The default setting is 0.2 milliseconds, meaning that if a loop requires fewer than 0.2 milliseconds to execute in parallel, it should be serialized.

Typically, *num* is set to be equal to the parallelization overhead. If the computation in a parallelized loop is very small and the time taken to execute these loops is spent primarily in the setting up of parallelization, these loops should be executed sequentially for better performance.

seqthreshold=_num_
Specifies the time, in milliseconds, beyond which a loop that was previously serialized by the dynamic profiler should revert to being a parallel loop. The default setting is 5 milliseconds, meaning that if a loop requires more than 5 milliseconds to execute serially, it should be parallelized.

seqthreshold acts as the reverse of **parthreshold**.

OpenMP environment variables for parallel processing

OpenMP runtime options affecting parallel processing are set by specifying OMP environment variables. These environment variables use syntax of the form:

►►──*env_variable*──=──*option_and_args*────────────────────────────────►◄

If an OMP environment variable is not explicitly set, its default setting is used.

For information about the OpenMP specification, see: http://www.openmp.org.

OMP_SCHEDULE=_algorithm_ **environment variable:** The OMP_SCHEDULE environment variable specifies the scheduling algorithm used for loops not explicitly assigned a scheduling algorithm with the **omp schedule** directive.

For example:
```
OMP_SCHEDULE="guided, 4"
```

Valid options for *algorithm* are:
- auto
- dynamic[, *n*]
- guided[, *n*]
- runtime
- static[, *n*]

If specifying a chunk size with *n*, the value of *n* must be an integer value of 1 or greater.

The default scheduling algorithm is **static**.

Related reference

"omp_set_schedule" on page 507

"omp_get_schedule" on page 507

OMP_NESTED: The OMP_NESTED=TRUE | FALSE environment variable enables or disables nested parallelism. Its setting can be overridden by calling the **omp_set_nested** runtime library function.

If nested parallelism is disabled, nested parallel regions are serialized and run in the current thread.

In the current implementation, nested parallel regions are always serialized. As a result, OMP_SET_NESTED does not have any effect, and **omp_get_nested** always returns 0. If **-qsmp=nested_par** option is on (only in non-strict OMP mode), nested parallel regions might employ additional threads as available. However, no new team is created to run nested parallel regions.

The default value for OMP_NESTED is FALSE.

OMP_DYNAMIC=TRUE | FALSE environment variable: The OMP_DYNAMIC environment variable enables or disables dynamic adjustment of the number of threads available for running parallel regions.

If it is set to TRUE, the number of threads available for executing parallel regions can be adjusted at run time to make the best use of system resources. For more information, see the description for **profilefreq=***num* in "XLSMPOPTS" on page 25.

If it is set to FALSE, dynamic adjustment is disabled.

The default setting is TRUE.

OMP_NUM_THREADS: The OMP_NUM_THREADS=*num* environment variable gives you full control over the number of user threads used to run your program. Some applications cannot use more threads than the maximum number of processors available. Other applications can experience significant performance improvements if they use more threads than there are processors.

num represents the number of parallel threads requested, which is usually equivalent to the number of processors available on the system. This number can be overridden by calling the **omp_set_num_threads** runtime library function.

The default value for *num* is the number of processors available on the system.

You can override the setting of OMP_NUM_THREADS for a given parallel section by using the **num_threads** clause available in several **#pragma omp** directives.

OMP_WAIT_POLICY environment variable:

The OMP_WAIT_POLICY environment variable gives hints to the compiler about the preferred behavior of waiting threads during program run time. The OMP_WAIT_POLICY environment variable sets the *wait-policy-var* internal control variable value.

The syntax is as follows:

```
►►──OMP_WAIT_POLICY=──┬─PASSIVE─┬─────────────────────────────────────────►◄
                      └─ACTIVE──┘
```

The default value for OMP_WAIT_POLICY is PASSIVE.

Use ACTIVE if you want waiting threads to be mostly active. With ACTIVE, the thread consumes processor cycles while waiting, if possible.

Use PASSIVE if you want waiting threads to be mostly passive. That is, the preference is for the thread to not consume processor cycles while waiting. For example, you prefer waiting threads to sleep or to yield the processor to other threads.

Note: If the OMP_WAIT_POLICY environment variable is set and the SPINS, YIELDS, or DELAYS suboptions of the XLSMPOPTS environment variable are specified, OMP_WAIT_POLICY takes precedence.

OMP_STACKSIZE environment variable:

The OMP_STACKSIZE environment variable indicates the stack size of threads created by the OpenMP run time. OMP_STACKSIZE sets the value of the *stacksize-var* internal control variable. OMP_STACKSIZE does not control the stack size of the master thread. The syntax is as follows:

```
►►──OMP_STACKSIZE=─────size─────────────────────────────────────────────►◄
```

By default, the size value is represented in Kilobytes. You can also use the suffixes B, K, M, or G if you want to indicate the size in Bytes, Kilobytes, Megabytes, or Gigabytes respectively. White space is allowed between and around the size value and the suffix. For example, these two examples both indicate a stack size of 10 Megabytes.
```
setenv OMP_STACKSIZE 10M
setenv OMP_STACKSIZE " 10 M "
```

If OMP_STACKSIZE is not set, the initial value of the *stacksize-var* internal control variable is set to the default value. The default value for 32-bit mode is 256M. For 64-bit mode, the default is up to the limit imposed by system resources. If the compiler cannot use the stack size specified or if OMP_STACKSIZE does not conform to the correct format, the compiler sets the environment variable to the default value. If the STACK suboption of the XLSMPOPTS environment variable and the OMP_STACKSIZE environment are specified, the OMP_STACKSIZE environment variable takes precedence.

OMP_THREAD_LIMIT environment variable:

Use OMP_THREAD_LIMIT to set the *thread-limit-var* internal control variable. *thread-limit-var* is used to indicate the number of OpenMP threads to be used for the whole program. The function **omp_get_thread_limit** can be used to retrieve this value at run time. The value for OMP_THREAD_LIMIT is a positive integer. If a value is chosen that is more than the number of threads that can be supported or

is not a positive integer, the runtime sets a default value for *thread-limit-var* of OMP_NUM_THREADS or the number of available processors, whichever is greater.

Note: if *thread-limit-var* is set, the default value of the *nthreads-var* internal control variable is equal to *thread-limit-var* or the number of available processors, whichever is less.

►►──OMP_THREAD_LIMIT=n──►◄

OMP_MAX_ACTIVE_LEVELS environment variable:

Use OMP_MAX_ACTIVE_LEVELS to set the *max-active-levels-var* internal control variable. This controls the maximum number of active nested parallel regions. In programs where nested parallelism is disabled, the initial value should be 1. In programs where nested parallelism is enabled, the initial value should be greater than 1. The function **omp_get_max_active_levels** can be used to retrieve this value at run time. The value for OMP_MAX_ACTIVE_LEVELS is a positive integer. If a positive integer is not specified, the default value for *max-active-levels-var* is set by the runtime.

►►──OMP_MAX_ACTIVE_LEVELS=n───►◄

Using custom compiler configuration files

The XL C/C++ compiler generates a default configuration file /opt/ibmcmp/vac/11.1/etc/vac.cfg.*$OSRelease.gcc$gccVersion*. For example, /opt/ibmcmp/vac/11.1/etc/vac.cfg.sles11.gcc432 or /opt/ibmcmp/vac/11.1/etc/vac.cfg.rhel5.5.gcc412 at installation time. (See the *XL C/C++ Installation Guide* for more information on the various tools you can use to generate the configuration file during installation.) The configuration file specifies information that the compiler uses when you invoke it.

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

If you want users to be able to choose among several sets of compiler options, you might want to use custom configuration files for specific needs. For example, you might want to enable **-qlist** by default for compilations using the **xlc** compiler invocation command. This is to avoid forcing your users to specify this option on the command line for every compilation. Because **-qlist** is automatically in effect every time the compiler is called with the **xlc** command.

You have several options for customizing configuration files:
- You can directly edit the default configuration file. In this case, the customized options will apply for all users for all compilations. The disadvantage of this option is that you will need to reapply your customizations to the new default configuration file that is provided every time you install a compiler update.
- You can use the default configuration file as the basis of customized copies that you specify at compile time with the **-F** option. In this case, the custom file overrides the default file on a per-compilation basis.

Note: This option requires you to reapply your customization after you apply service to the compiler.

- You can create custom, or user-defined, configuration files that are specified at compile time with the XLC_USR_CONFIG environment variable. In this case, the custom user-defined files complement, rather than override, the default configuration file, and they can be specified on a per-compilation or global basis. The advantage of this option is that you do not need to modify your existing, custom configuration files when a new system configuration file is installed during an update installation. Procedures for creating custom, user-defined configuration files are provided below.

Related information:
- "-F" on page 133
- "Compile-time and link-time environment variables" on page 24

Creating custom configuration files

If you use the XLC_USR_CONFIG environment variable to instruct the compiler to use a custom user-defined configuration file, the compiler examines and processes the settings in that user-defined configuration file before looking at the settings in the default system configuration file.

To create a custom user-defined configuration file, you add stanzas which specify multiple levels of the **use** attribute. The user-defined configuration file can reference definitions specified elsewhere in the same file, as well as those specified in the system configuration file. For a given compilation, when the compiler looks for a given stanza, it searches from the beginning of the user-defined configuration file and follows any other stanza named in the use attribute, including those specified in the system configuration file.

If the stanza named in the **use** attribute has a name different from the stanza currently being processed, the search for the use stanza starts from the beginning of the user-defined configuration file. This is the case for stanzas A, C, and D which you see in the following example. However, if the stanza in the **use** attribute has the same name as the stanza currently being processed, as is the case of the two B stanzas in the example, the search for the **use** stanza starts from the location of the current stanza.

The following example shows how you can use multiple levels for the **use** attribute. This example uses the **options** attribute to help show how the **use** attribute works, but any other attributes, such as **libraries** can also be used.

```
A: use =DEFLT
    options=<set of options A>
B: use =B
    options=<set of options B1>
B: use =D
    options=<set of options B2>
C: use =A
    options=<set of options C>
D: use =A
    options=<set of options D>
DEFLT:
    options=<set of options Z>
```

Figure 1. Sample configuration file

In this example:

- stanza A uses option sets *A* and *Z*
- stanza B uses option sets *B1*, *B2*, *D*, *A*, and *Z*
- stanza C uses option sets *C*, *A*, and *Z*
- stanza D uses option sets *D*, *A*, and *Z*

Attributes are processed in the same order as the stanzas. The order in which the options are specified is important for option resolution. Ordinarily, if an option is specified more than once, the last specified instance of that option wins.

By default, values defined in a stanza in a configuration file are added to the list of values specified in previously processed stanzas. For example, assume that the XLC_USR_CONFIG environment variable is set to point to the user-defined configuration file at ~/userconfig1. With the user-defined and default configuration files shown in the following example, the compiler references the **xlc** stanza in the user-defined configuration file and uses the option sets specified in the configuration files in the following order: *A1*, *A*, *D*, and *C*.

```
xlc:  use=xlc
    options= <A1>

DEFLT: use=DEFLT
    options=<D>
```

```
xlc:  use=DEFLT
    options=<A>

DEFLT:
    options=<C>
```

Figure 2. Custom user-defined configuration file ~/userconfig1

Figure 3. Default configuration file vac.cfg

Overriding the default order of attribute values

You can override the default order of attribute values by changing the assignment operator(=) for any attribute in the configuration file.

Table 8. Assignment operators and attribute ordering

Assignment Operator	Description
-=	Prepend the following values before any values determined by the default search order.
:=	Replace any values determined by the default search order with the following values.

Table 8. Assignment operators and attribute ordering (continued)

Assignment Operator	Description
+=	Append the following values after any values determined by the default search order.

For example, assume that the XLC_USR_CONFIG environment variable is set to point to the custom user-defined configuration file at ~/userconfig2.

Custom user-defined configuration file
~/userconfig2   **Default configuration file vac.cfg**

```
xlc_prepend: use=xlc              xlc: use=DEFLT
            options-=<B1>               options=<B>
xlc_replace: use=xlc
            options:=<B2>         DEFLT:
xlc_append: use=xlc                     options=<C>
            options+=<B3>

DEFLT: use=DEFLT
     options=<D>
```

The stanzas in the preceding configuration files use the following option sets, in the following orders:

1. stanza xlc uses *B*, *D*, and *C*
2. stanza xlc_prepend uses *B1*, *B*, *D*, and *C*
3. stanza xlc_replace uses *B2*
4. stanza xlc_append uses *B*, *D*, *C*, and *B3*

You can also use assignment operators to specify an attribute more than once. For example:

```
xlc:
    use=xlc
    options-=-Isome_include_path
    options+=some options
```

Figure 4. Using additional assignment operations

Examples of stanzas in custom configuration files

`DEFLT: use=DEFLT` `   options = -g`	This example specifies that the **-g** option is to be used in all compilations.

`xlc: use=xlc`
`options+=-qlist`
`xlc_r: use=xlc_r`
`options+=-qlist`
	This example specifies that **-qlist** is to be used for any compilation called by the **xlc** and **xlc_r** commands. This **-qlist** specification overrides the default setting of **-qlist** specified in the system configuration file.

`DEFLT: use=DEFLT` `   libraries=-L/home/user/lib,-lmylib`	This example specifies that all compilations should link with /home/user/lib/libmylib.a.

Configuring the gxlc or gxlc++ option mapping

The **gxlc** and **gxlc++** utilities use the configuration file /opt/ibmcmp/vac/11.1/ etc/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/C++ option, and a string for the XL C/C++ option. The three fields must be separated by white space. 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** or **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 the utility 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"
```

      The utility will then expect a value for these options.

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

      n   Tells the utility to process the option listed in the *gcc_or_g++_option* field

      i   Tells the utility to ignore the option listed in the *gcc_or_g++_option* field. The utility will generate a message that this has been done, and continue processing the given options.

      e   Tells the utility to halt processing if the option listed in the *gcc_or_g++_option* field is encountered. The utility will also generate an error message.

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

```
nni*      "-I-"
```

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

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

c   Tells the utility to translate the option only for C.

x   Tells the utility to translate the option only for C++.

*   Tells **gxlc** or **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 GNU C/C++ option. 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, the utility 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/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 **-isystem** option is not supported and uses a parameter. Both must be ignored by the application. In this case, the entry is:

```
nni*        "-isystem %"
```

For a complete list of GNU C and C++ and XL C/C++ option mappings, refer to:

http://www.ibm.com/support/docview.wss?uid=swg27011888

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

Chapter 3. Tracking and reporting compiler usage

You can use the utilization tracking and reporting feature to record and analyze which users in your organization are using the compiler and the number of users using it concurrently. This information can help you determine whether your organization's use of the compiler exceeds your compiler license entitlements.

To use this feature, follow these steps:

1. Understand how the feature works. See "Understanding utilization tracking and reporting" for more information.
2. Investigate how the compiler is used in your organization, and decide how you track the compiler usage accordingly. See "Preparing to use this feature" on page 48 for more information.
3. Configure and enable utilization tracking. See "Configuring utilization tracking" on page 53 for more information.
4. Use the utilization reporting tool to generate usage reports or prune usage files. See "Generating usage reports" on page 61 or "Pruning usage files" on page 64 for more information.

Understanding utilization tracking and reporting

The utilization tracking and reporting feature provides a mechanism for you to detect whether your organization's use of the compiler exceeds your compiler license entitlements. This section introduces the feature, describes how it works, and illustrates its typical usage scenarios.

Overview

When utilization tracking is enabled, all compiler invocations are recorded in a file. This file is called a usage file and it has the `.cuf` extension. You can then use the utilization reporting tool to generate a report from one or more of these usage files, and optionally prune the usage files.

You can use the utilization tracking and reporting feature in various ways based on how the compiler is used in your organization. "Four usage scenarios" on page 40 illustrates the typical usage scenarios of this feature.

The following sections introduce the configuration of the utilization tracking functionality and the usage of the utilization reporting tool.

Utilization tracking

A utilization tracking configuration file `urtxlc_cpp1101linux.cfg` is included in the default compiler installation. You can use this file to enable utilization tracking and control different aspects of the tracking.

A symlink `urt_client.cfg` is also included in the default compiler installation. It points to the location of the utilization tracking configuration file. If you want to put the utilization tracking configuration file in a different location, you can modify the symlink accordingly.

For more information, see "Configuring utilization tracking" on page 53.

Note: Utilization tracking is disabled by default.

Utilization reporting tool

The utilization reporting tool generates compiler usage reports based on the information in the usage files. You can optionally prune the usage files with the tool. For more information, see "Generating usage reports" on page 61 and "Pruning usage files" on page 64.

Four usage scenarios

This section describes four possible scenarios for managing the compiler usage, for recording the compiler usage information and for generating reports from this information.

The following scenarios describe some typical ways that your organization might be using the compiler and illustrates how you can use this feature to track compiler usage in each case.

Note: Actual usage is not limited to these scenarios.

"Scenario: One machine, one shared .cuf file"

"Scenario: One machine, multiple .cuf files" on page 42

"Scenario: Multiple machines, one shared .cuf file" on page 44

"Scenario: Multiple machines, multiple .cuf files" on page 46

Scenario: One machine, one shared .cuf file

This scenario describes an environment where all the compilations are done on one machine and all users share one `.cuf` file.

The advantage of using the approach in this scenario is that it simplifies report generation and usage file pruning, because the utilization report tool only need to access one `.cuf` file. The disadvantage is that all compiler users need to compete for access to this file. Because the file might become large, it might have an impact on performance. Some setup work is also required to create the shared `.cuf` file and to give all compiler users write access. The "The number of usage files" on page 51 section provides detailed information about using a single usage file for all compiler users.

In this scenario, two compiler users run the compiler on the same machine and their utilization information is recorded in a shared `.cuf` file. The utilization tracking configuration file for the compiler is modified to point to the location of the `.cuf` file. When the compiler is invoked, it writes the utilization information to that file. You can then use the utilization reporting tool to retrieve the utilization information from the file and generate usage reports.

The following diagram illustrates this scenario.

Utilization tracking

1

User: user1 — Invoke the compiler → Compiler — Write to file in /xyz

Read

Utilization tracking configuration file

.cuf

Read

1

User: user2 — Invoke the compiler → Compiler — Write to file in /xyz

Utilization reporting

Read report → Report

2

User: user3

Generate

3 Invoke urt with -qusagefileloc=/xyz

urt — Read

Read

urt configuration file
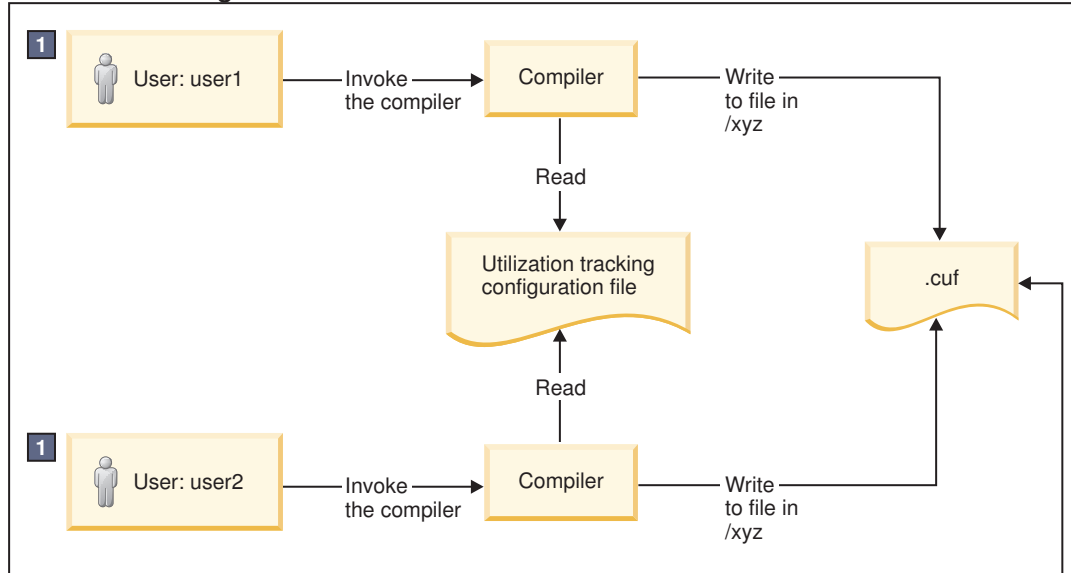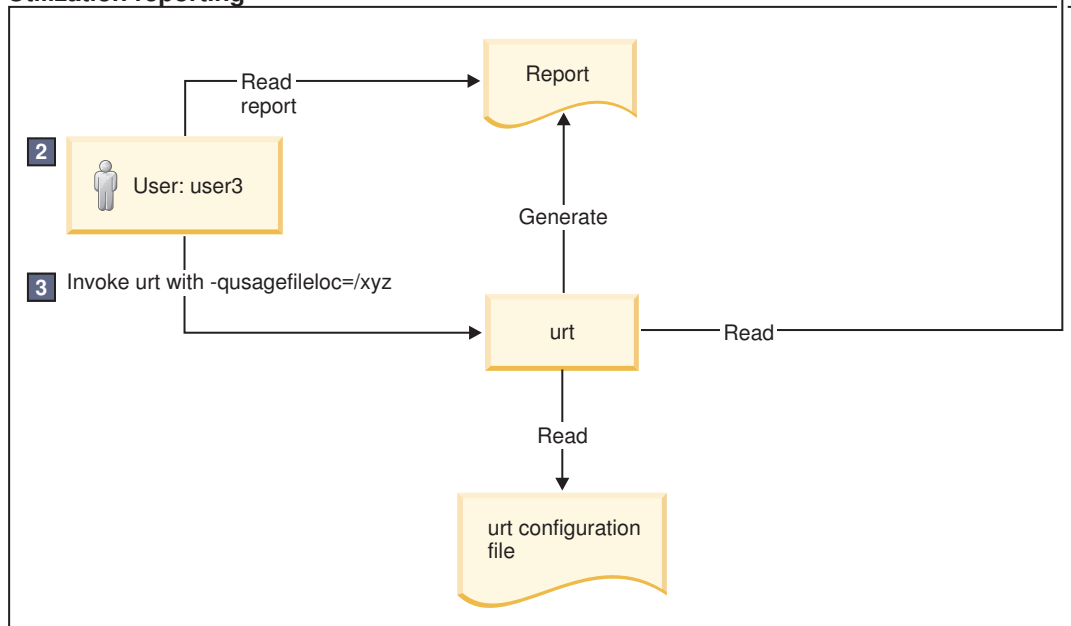
1. Both user1 and user2 need write access to the `.cuf` file in /xyz.

2. user3 needs read access to the `.cuf` file in/xyz to generate the usage report, and write access to prune the `.cuf` file.

3. A cron job can be created to run **urt** automatically on a regular basis.

Figure 5. Compiler users use a single machine, with a shared .cuf file

The diagram reflects the following points:

1. user1 and user2 use the same utilization tracking configuration file, which manages the tracking functionality centrally. A common location `/xyz` is created to keep a shared `.cuf` file.

2. When user1 and user2 invoke the compiler, the utilization information is recorded in the `.cuf` file under the common directory `/xyz`.

3. user3 invokes **urt** with `-qusagefileloc=/xyz` to generate usage reports.

Note: Regular running of the utilization reporting tool can prevent these files from growing too big, because you can prune the usage files with this tool.

Scenario: One machine, multiple .cuf files

This scenario describes an environment where all the compilations are done on one machine and all users have their own `.cuf` files.

The approach in this scenario has the following advantages:

- Compiler users do not have to compete for access to a single `.cuf` file, and this might result in better performance.
- You do not need to set up write access to a single common location for all compiler users. They already have write access to their own home directories.

However, using multiple `.cuf` files that are automatically created in each user's home directory might have the following issues:

- Compiler users might not know that the file has been created or what it is when they see the file. In this case, they might delete the file.
- Some users' home directories might be on file systems that are mounted from a remote system. This causes utilization tracking to use a remote file, which might affect performance.
- Compiler users might not want `.cuf` files to take up space in their /home directories.

Instead of using each user's home directory, the `.cuf` files for each user can be created in a common location. The "Usage file location" on page 50 section provides detailed information about how to create these files in a common location.

In this scenario, two compiler users run the compiler on the same machine and they have their own `.cuf` files. When the compiler is invoked, it automatically creates a `.cuf` file for each user and writes the utilization information to that file. You can then use the utilization reporting tool to retrieve the utilization information from the file and generate usage reports.

The following diagram illustrates this scenario.

Figure 6. Compiler users use one machine, with separate .cuf files
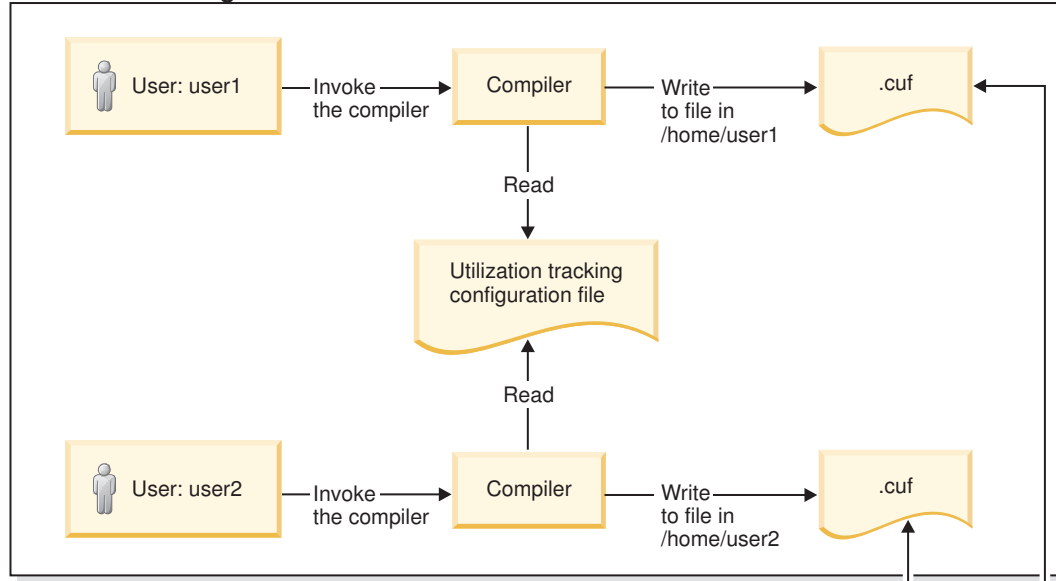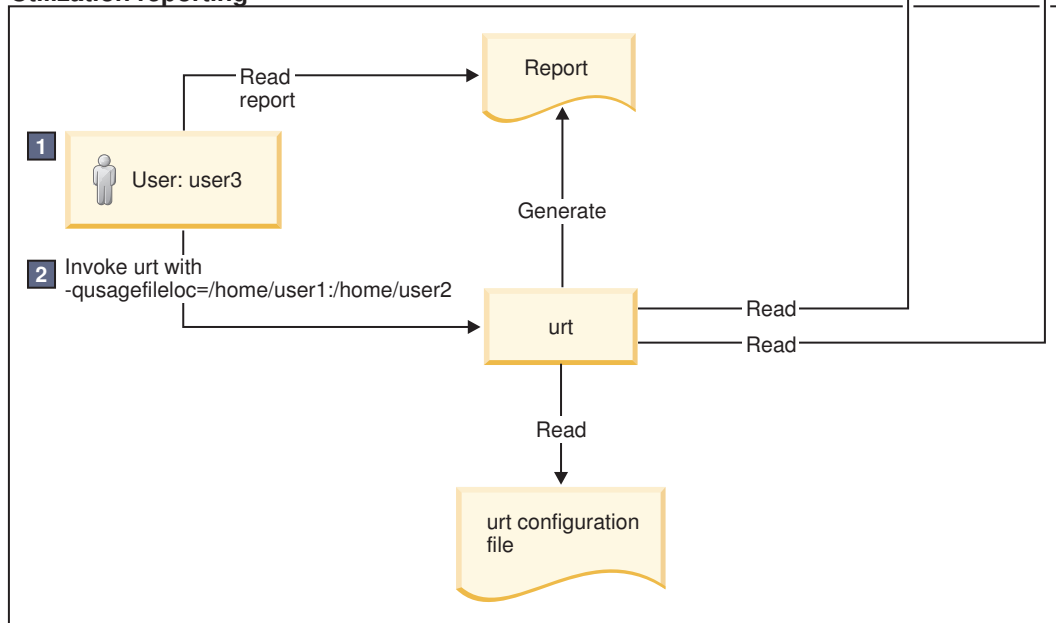
1. user3 needs read access to `.cuf` files in `/home/user1` and `/home/user2` to generate the usage report, and write access to prune the usage files.

2. A cron job can be created to run **urt** automatically on a regular basis.

This diagram reflects the following points:

1. user1 and user2 use the same utilization tracking configuration file, which manages the tracking functionality centrally.

2. When user1 and user2 invoke the compiler, the utilization information is recorded in the two `.cuf` files under their respective home directories, `/home/user1` and `/home/user2`.

3. user3 invokes **urt** with `-qusagefileloc=/home/user1:/home/user2` to generate usage reports.

Note: If you need to find out which home directories contain usage files, you can invoke **urt** as follows:

```
urt -qusagefileloc=/home -qmaxsubdirs=1
```

In this case, **urt** looks for `.cuf` files in all users' home directories.

Scenario: Multiple machines, one shared .cuf file

This scenario describes an environment where the compilations are done on multiple machines but all users share a single `.cuf` file.

The advantage of the approach in this scenario is that using one `.cuf` file can simplify the report generation and the usage file pruning process. The section "The number of usage files" on page 51 provides detailed information about using a single usage file for all compiler users. The `.cuf` file is already on the machine where the utilization reporting tool is installed. You do not need to copy the file to that machine or install the tool on multiple machines to prune the `.cuf` files.

This approach has the following disadvantages:
* The compiler users must compete for access to one usage file. Because the file might become large, it might have an impact on performance.
* Some setup work is required to create the shared `.cuf` file and to give all compiler users write access on a network file system.
* The efficiency of the whole process depends on the speed and reliability of the network file system, because the compilers and the `.cuf` file are on different machines. For example, some file systems are better than others in supporting file locking, which is required for concurrent access by multiple users.

In this scenario, two compiler users run the compilers on separate machines and they use one shared `.cuf` file on a network file system, such as NFS, DFS, or AFS. When the compiler is invoked, it writes the utilization information to that file. You can then use the utilization reporting tool to retrieve the utilization information from the file and generate usage reports.

The following diagram illustrates this scenario.
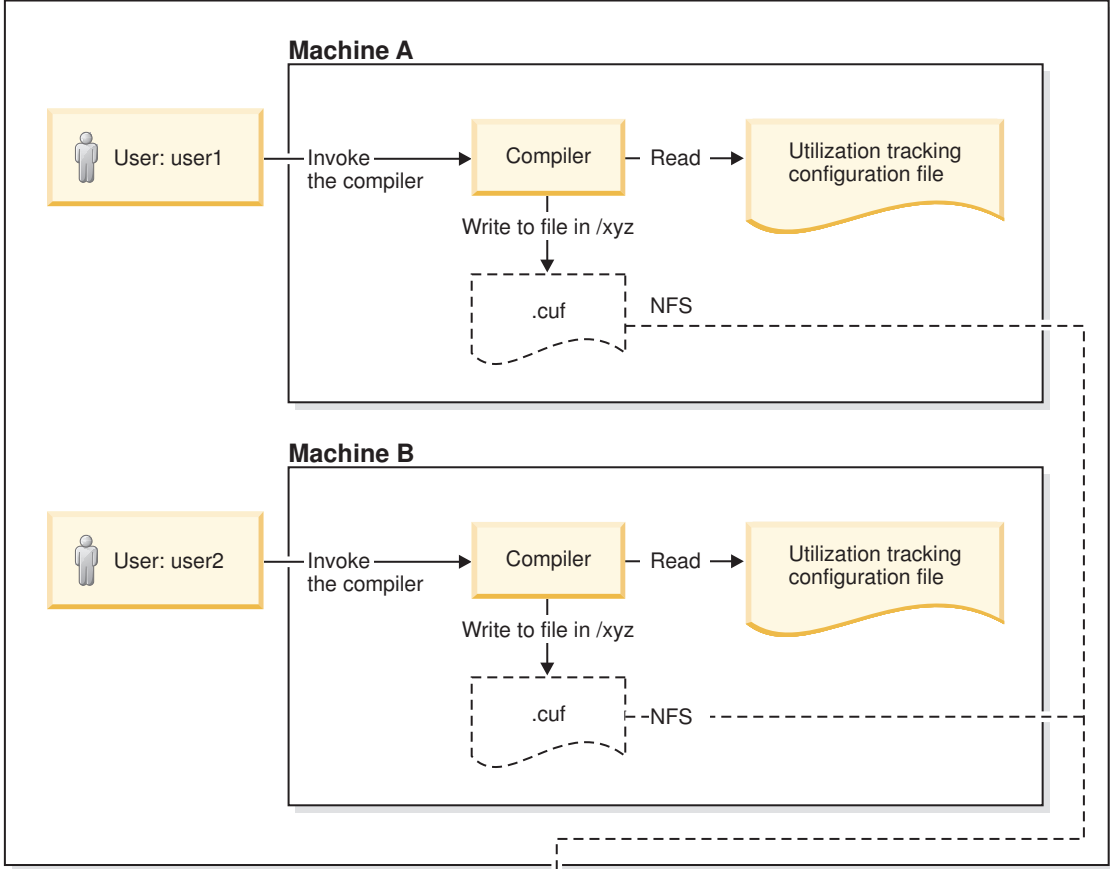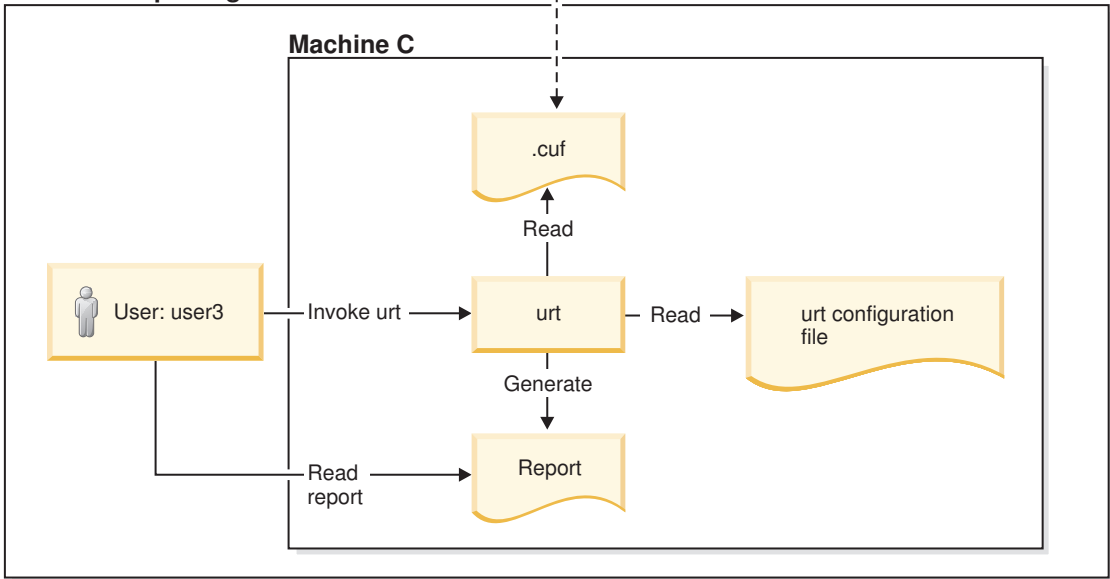
1. On Machine A and Machine B, mount point /xyz is created to Machine C. All compiler utilization is recorded in the .cuf file, from which the usage report is generated.

Figure 7. Compiler users use multiple machines, with a shared .cuf file

This diagram reflects the following points:

1. Utilization tracking is configured respectively on Machine A and Machine B.

Notes:

- Although each machine has its own configuration file, the contents of these files must be the same.
- Centrally managing the utilization tracking functionality can reduce your configuration effort and eliminate possible errors. The "Central configuration" on page 49 section provides detailed information about how you can use a common configuration file shared by compiler users using different machines.

2. A network file system is set up for the central management of the `.cuf` files. When user1 and user2 invoke the compilers from Machine A and Machine B, the utilization information of both compilers is written to the `.cuf` file on Machine C.

3. user3 invokes **urt** to generate usage reports from the `.cuf` file on Machine C.

Note: You can use the utilization reporting tool to prune the usage files regularly to prevent them from growing too big.

Scenario: Multiple machines, multiple .cuf files

This scenario describes an environment where the compilations are done on multiple machines and all users have their own usage files.

In this scenario, two compiler users run the compilers on separate machines and they have their own `.cuf` files. When the compiler is invoked, it writes the utilization information to that file. You can then use the utilization reporting tool to retrieve the utilization information from the file and generate usage reports. This tool can be run on either of the machines on which the compiler is installed or on a different machine.

Note: The utilization reporting tool requires access to all the `.cuf` files.
You can use either of the following methods to make the files accessible in this example:

- Use a network file system, such as NFS, DFS, or AFS.
- Copy the files from their original locations to the machine where you plan to run the utilization reporting tool. You can use **ftp**, **rcp**, **rsync** or any other remote copy command to copy the files.

The following diagram illustrates this scenario.

Figure 8. Compiler users use multiple machines, with multiple .cuf files

Utilization tracking

Machine A

User: user1 — Invoke the compiler → Compiler — Read → Utilization tracking configuration file

Write to file in home/user1 ↓ .cuf — Copy

Machine B

User: user2 — Invoke the compiler → Compiler — Read → Utilization tracking configuration file

Write to file in /home/user2 ↓ .cuf — Copy

Utilization reporting

Machine C

.cuf

Read ↑

User: user3 — Invoke urt → urt — Read → urt configuration file

Generate ↓

Read report → Report
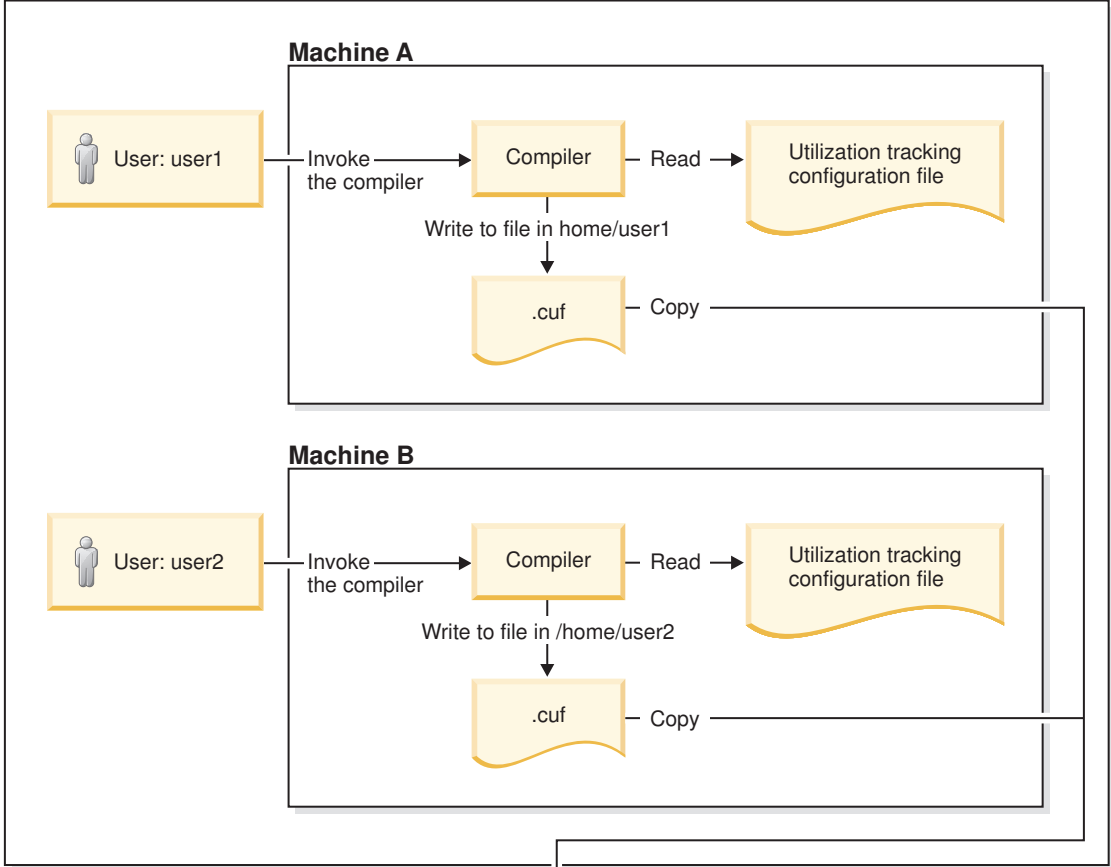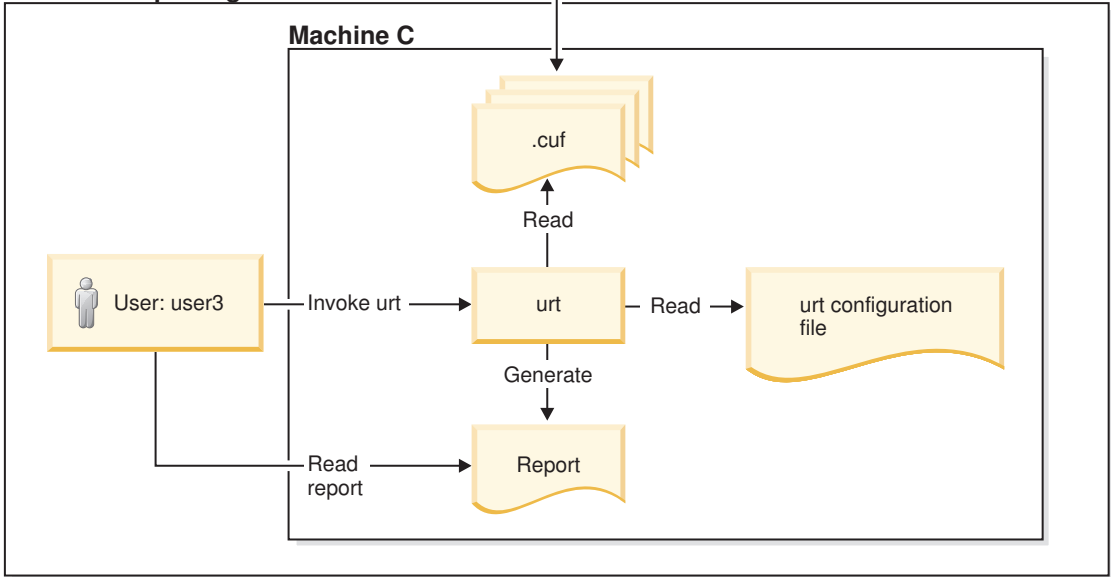
1. user3 copies the `.cuf` files to Machine C. A cron job can be created to copy the files automatically on a regular basis.

This diagram reflects the following points:

1. Utilization tracking is configured respectively on Machine A and Machine B.

Notes:
- Although each machine has its own configuration file, the contents of these files must be the same.
- Centrally managing the utilization tracking functionality can reduce your configuration effort and eliminate possible errors. The "Central configuration" on page 49 section provides detailed information about how you can use a common configuration file shared by compiler users using different machines.

2. When user1 and user2 invoke the compilers, the utilization information is recorded in the two `.cuf` files under their respective home directories, `/home/user1` and `/home/user2`.

   Note: These `.cuf` files can also be created in another common location, for example, `/var/tmp`. The "Usage file location" on page 50 section provides detailed information about how to create these files in a common location.

3. user3 copies the two `.cuf` files from Machine A and Machine B to Machine C.

4. user3 invokes **urt** to generate usage reports from the `.cuf` files on Machine C.

Related information
- "Preparing to use this feature"
- "Configuring utilization tracking" on page 53
- "Generating usage reports" on page 61
- "Pruning usage files" on page 64

Preparing to use this feature

Before enabling utilization tracking within your organization, you must consider certain factors related to how the compiler is used in your organization.

The following sections describe those considerations in detail:

Time synchronization

If you plan to track the utilization of the compiler on more than one machine, you must consider synchronizing the time across the machines.

The usage report generated by the utilization reporting tool lists the time when the compiler invocations start and end. The report also determines which invocations are concurrent. This information is much less reliable and useful if time is not synchronized across these machines.

If you are unable to synchronize time across different machines, you can use the **-qadjusttime** option to instruct the utilization reporting tool to adjust the times that have been recorded.

License types and user information

Before you start to use this feature, you need the number and type of license entitlements for your organization.

The license and user information that you need is as follows:
- The number of Concurrent User licenses that you have for this compiler. This information is required for the **-qmaxconcurrentusers** entry in the utilization tracking configuration file.

- The users who have their own Authorized User license for this compiler. This information is used for the **-qexemptconcurrentusers** entry in the utilization tracking configuration file.
- The users who use the compiler with multiple accounts. This information is used for the **-qsameuser** option for the utilization reporting tool.

Note: It is not mandatory to specify the users who have their own Authorized User license and the users who use the compiler with multiple accounts, but it improves the accuracy of the usage reports generated by the utilization reporting tool. For detailed information, see "Concurrent user considerations."

Central configuration

Configuring utilization tracking the same for all compiler users is very important, because it can ensure the accuracy of your utilization tracking, and minimize your configuration and maintenance effort. You can achieve this by ensuring that all users use the same utilization tracking configuration file.

If you have only one installation of the compiler, you can directly edit the utilization tracking configuration file. Every compiler user can automatically use that configuration file.

If you have multiple installations of the compiler, you need to maintain a single utilization tracking config file and reference it from all installations. Any changes you make to the utilization tracking configuration file, including enabling or disabling utilization tracking, can automatically apply to all compiler installations when users invoke the compiler. In each installation, there is a symlink named `urt_client.cfg`, located under /opt/ibmcmp/vacpp/11.1/urt. Modify the symlink to point to this shared instance of the configuration file.

If the compiler is installed on multiple machines, the utilization tracking configuration file needs to be placed on a network file system, such as NFS, DFS, or AFS, to be used by the compiler on each machine.

Note: If it is not possible for you to use a single utilization tracking configuration file for all compiler users, you must ensure all utilization tracking configuration files for each compiler installation are consistent. Using different configurations for the same compiler is not supported.

Concurrent user considerations

Invocations of the compiler are considered concurrent when their start time and end times overlap. This section provides the information about how the utilization reporting tool counts concurrent users and the ways to increase the accuracy of the usage reports.

When the utilization reporting tool counts concurrent users, it looks at the user account information that has been captured in the usage files. The account information consists of a user name, a user ID, and a host name. By default, each unique combination of this account information is considered and counted as a different user. However, invocations of the compiler by the following users must not be included in the count of concurrent users:

- Users who have their own Authorized User license are considered exempt users, because their use of the compiler does not consume any Concurrent User licences.

- Users who have multiple accounts. Because the accounts belong to the same user, invocations of the compiler while logged on using those accounts are counted as usage by a single user.

The utilization reporting tool can account for the above situations if you provide it with information regarding exempt users and users with multiple accounts. Here is how you can provide the information:

- Specify the **-qexemptconcurrentusers** entry in the utilization tracking configuration file. This entry specifies users with Authorized User licenses.
- Specify the **-qsameuser urt** command-line option. This option specifies users with multiple accounts.

Notes:

- When the number of concurrent users is adjusted with **-qexemptconcurrentusers** or **-qsameuser**, the utilization reporting tool generates a message to indicate that the concurrent usage information is adjusted.
- The number of concurrent users might be zero if all concurrent invocations are invoked by exempt users. The tool also generates a message with this information.

Usage file considerations

Usage (`.cuf`) files are used to store compiler usage information. This section provides information that helps you decide how you want to generate and use these files.

Usage file location

Usage files can be created in each user's home directory, or they can be created in a central location for all users.

With utilization tracking enabled, when a compiler user compiles a program, a `.cuf` file is automatically created in the user's home directory in case the file does not exist. This is convenient for testing the utilization tracking feature because users already have write access to their own home directories, which means no additional setup is required. However, this might have the following issues:

- Compiler users might not know that the file has been created or what it is when they see the file. In this case, they might delete the file.
- Some users' home directories might be on file systems that are mounted from a remote system. This causes utilization tracking to use a remote file, which might affect performance.
- Compiler users might not want usage files to take up space in the `/home` directory.

A good alternative is to set up a central location where the usage files can be created, and provide appropriate access to that location for both the compiler users and the utilization reporting tool users. This can be set up by using the `other/world` permissions or by using group permissions.

For example, if the central location is a directory named `/var/tmp/track_compiler_use`, you can modify the **-qusagefileloc** entry in the utilization tracking configuration file as follows:

```
-qusagefileloc=/var/tmp/track_compiler_use/$LOGNAME.cuf
```

This creates a `.cuf` file for each user in the specified location, such as `user1.cuf` or `user2.cuf`. It is easier to run the utilization reporting tool to generate the usage

report from the `.cuf` files in this central location. You only need to pass the path of the location, `/var/tmp/track_compiler_use` to the utilization reporting tool , and then the tool can read all the `.cuf` files in that location.

If the compiler users are running the compiler on more than one machine, you need to add *$HOSTNAME* to the **-qusagefileloc** entry to ensure that there are no collisions in the file names. For example, you can specify the **-qusagefileloc** entry as follows:

```
-qusagefileloc=/var/tmp/track_compiler_use/$HOSTNAME_$LOGNAME.cuf
```

This creates a `.cuf` file for each user, and the name of that `.cuf` file also contains the name of the host on which the compiler is used, such as `host1_user1.cuf`.

The number of usage files

You can use one usage file or separate usage files for different compiler users.

Using separate usage files for different compiler users

The advantages of using separate usage files are as follows:
- It might provide better performance because compiler users access their own usage files instead of competing for access to a shared one and separate usage files are usually smaller.
- Usage file for a user can be automatically created when the user uses the compiler to compile a program. There is no need to explicitly create a usage file for each user beforehand. For more information, see "Usage file location" on page 50.
- When generating utilization reports, you usually include all compiler users. However, if there are circumstances in which you want to exclude some users, you can simply omit their usage files when you invoke the utilization reporting tool. For example, you might want to omit users who have their own Authorized User license.

The disadvantage is that you might have to maintain separate usage files for different users.

Using a single usage file for all compiler users

The advantage of using a shared usage file for all users is that you only need to maintain a single file instead of multiple files. However, with a single usage file, you lose the flexibility and possible performance benefits of using multiple usage files, as described in the preceding subsection.

The compiler provides an empty usage file `urtstub.cuf` in the `usr/vac/urt` directory. You can create a usage file for all compiler users by copying the empty usage file to a directory where they all have write access. In this case, you need to change the **-qusagefileloc** entry in the utilization tracking configuration file to point to the location of the usage file.

Usage files on multiple machines

If you use the compiler on multiple machines, you need to decide how to make the usage files available for the utilization reporting tool.

You can use various methods to make the usage files available for the utilization reporting tool to generate usage reports and prune the usage files. Choose one of the following approaches to manage usage files on multiple machines:

- Copy the usage files from the machines where the compiler is used to the machine where the utilization reporting tool is installed. You can use any remote copy command, for example, **ftp**, **rcp**, **scp**, and **rsync**. In this case, the usage files are being accessed locally by both the compiler, for utilization tracking, and by the utilization reporting tool, for generating the usage report. Accessing the files locally yields the best performance.
- Use a distributed file system to export the file system from the machines where the compiler is used, and mount those file systems on the machine where the utilization reporting tool is installed. When you run the utilization reporting tool, it can access the usage files remotely via the mounted file systems.
- You can also export the file system from the machine where the utilization reporting tool is installed, and mount that file system on each machine where the compiler is used, using it as the location of the usage files where the compiler is recording its utilization. In this approach, the compiler records utilization in a remote usage file, and the utilization reporting tool reads the usage file locally.

  Note: If you find this degrades the performance of the compiler, consider using one of the first two approaches instead.

Usage file size

You need to consider the fact that the size of the usage files might grow quickly, especially when you use a shared file for all compiler users. If the usage file gets too large, it might affect utilization tracking performance.

To keep the usage files from growing quickly, you can optionally prune the usage files when you generate usage reports. You can also prune the files regularly using cron.

For more information about how to prune files, see "Pruning usage files" on page 64.

Regular utilization checking

You can run the utilization reporting tool on a regular basis to verify whether the usage of the compiler is compliant with the Concurrent User licenses you have purchased. You can create a cron job to do this automatically.

If the usage files need to be copied to the machine where the utilization reporting tool is running, you can also automate the copying task with a cron job.

Another reason for running the tool regularly is to prune the usage files to control the size of these files.

Note: To reduce contention for read and write access to the usage files, run the utilization reporting tool or copy the usage files when the compiler is not being used.

Testing utilization tracking

Before you begin to track the compiler usage for all users in your organization, you can test the feature with a limited number of users or with a separate compiler installation. During this testing, you can try different configurations so as to decide the best setup for your organization.

Testing with a limited number of users

To enable compiler utilization tracking for a limited number of users, you can use a separate utilization tracking configuration file and ask only these users to use the file. Other users of the same installation use the default utilization tracking configuration file in which utilization tracking is disabled, and their use of the compiler is therefore not recorded.

The default compiler configuration file, vac.cfg.*$OSRelease*.gcc*$gccVersion*. For example, vac.cfg.sles11.gcc432 or vac.cfg.rhel5.5.gcc412 contains two entries, *xlurt_cfg_path* and *xlurt_cfg_name*, which specify the location of the utilization tracking configuration file. You need to perform the following tasks to let the specified users use the separate utilization tracking configuration file:

1. Create a separate compiler configuration file or stanza, in which the *xlurt_cfg_path* and *xlurt_cfg_name* entries specify the location of the utilization tracking configuration file you want to use.
2. Ask these users to use the following compiler option or environment variable to instruct the compiler to use the separate compiler configuration file or stanza, which in turn allows them to use the separate utilization tracking configuration file.
   - The **-F** option
   - The XLC_USR_CONFIG environment variable

Note: This approach is only for testing the utilization tracking feature. Do not use it for tracking all compiler utilization in your organization unless you can ensure that all compiler invocations are done with the **-F** option or the **XLC_USR_CONFIG** environment variable set.

Testing with a separate compiler installation

You can install a separate instance of the compiler for testing utilization tracking. In this case, you can directly modify the utilization tracking configuration file in that installation to enable and configure utilization tracking. The compiler users involved in the testing do not need to perform any task for the tracking.

When you are satisfied that you have found the best utilization tracking configuration for your organization, you can enable it for all compiler users in your organization.

Related information
- "Configuring utilization tracking"

Configuring utilization tracking

You can use the utilization tracking configuration file to enable and configure the utilization tracking functionality.

The default location of the configuration file is /opt/ibmcmp/vacpp/11.1/urt and its file name is urtxlc_cpp1101linux.cfg.

The compiler uses a symlink to specify the location of the utilization tracking configuration file. The symlink is also located in /opt/ibmcmp/vacpp/11.1/urt and its name is urt_client.cfg. In the following situations, you might need to change the symlink:

- If you want to use a utilization tracking configuration file in a different location, change the symlink to point to this location.
- If you have multiple installations of the same compiler, and you plan to use a single utilization tracking configuration file, change the symlink in each installation to point to that file. For more information, see "Central configuration" on page 49.

Note: Installing a PTF update does not overwrite the utilization tracking configuration file.

You can use the entries in the utilization tracking configuration file to configure many aspects of the way compiler usage is tracked. For details about the specific entries in that file and how they can be modified, see "Editing utilization tracking configuration file entries."

Editing utilization tracking configuration file entries

You can configure different aspects of utilization tracking by editing the entries in the utilization tracking configuration file.

The entries are divided into two categories.

1. The entries in the *Product information* category identify the compiler. Do not modify these entries.
2. The entries in the *Tracking configuration* category can be used to configure utilization tracking for this product. Changes to these entries take effect in the usage file upon the next compiler invocation. In this case, the compiler emits a message to indicate that the new configuration values have been saved in the usage file. When you generate a report from the usage file, the new values are used.

The following rules apply when you modify the entries:
- The following entries are written to the usage files whenever you change them, and they are used the next time the utilization reporting tool generates a report from the usage files. These configuration entries must be the same for all compiler users.
    - **-qmaxconcurrentusers**
    - **-qexemptconcurrentusers**
    - **-qqualhostname**
- If **-qqualhostname** is changed, you must discard any existing usage files and start tracking utilization again with new usage files. Otherwise some invocations are recorded with qualified host names and some are recorded with unqualified host names.

Notes:
- The entries are not compiler options. They can only be used in the utilization tracking configuration file.
- If the **-qexemptconcurrentusers** entry is specified multiple times in the utilization tracking configuration file, all the specified instances are used. If other entries are specified multiple times, the value of the last one overrides previous ones.
- The compiler generates a message if you specify the above entries with different values for different users when using more than one utilization tracking configuration file. You must modify the entries to keep them consistent, or make sure all compiler users use a single utilization configuration file.

Product information

-qprodId=_product_identifier_string_
  Indicates the unique product identifier string.

-qprodVer=_product_version_
  Indicates the product version.

-qprodRel=_product_release_
  Indicates the product release.

-qprodName=_product_name_
  Indicates the product name.

-qconcurrentusagescope=prod | ver | rel
  Specifies the level at which concurrent users are counted, and their numbers are limited. The suboptions are as follows:
  - **prod** indicates the product level.
  - **ver** indicates the version level.
  - **rel** indicates the release level.

  Default: **-qconcurrentusagescope=prod**

Tracking configuration

-qmaxconcurrentusers=_number_

  Specifies the maximum number of concurrent users. It is the number of Concurrent User license that you have purchased for the product. When the utilization reporting tool generates a report from the usage file, it determines whether your compiler usage in your organization has exceeded this maximum number of concurrent users.

  Note: You must update this entry to reflect the actual number of Concurrent User licenses that you have purchased.

  Default: 0

-qexemptconcurrentusers ="_user_account_info_1_ **[|** _user_account_info_2_ **| ... |** _user_account_info_n_**]"**

  Specify exempt users who have their own Authorized User license. Exempt users can have as many concurrent invocations of the compiler as they want, without affecting the number of Concurrent User licenses available in your organization. When the utilization reporting tool generates a usage report, it does not include such users in the count of concurrent users.

  _user_account_info_ can be any combination of the following items:
  - name(_user_name_)
  - uid(_user_ID_)
  - host(_host_name_)

  Users whose information matches the specified criteria are considered exempt users. For example, to indicate that _user1@host1_ and _user2@host1_ are exempt users, you can specify this entry in either of the following forms:
  - `-qexemptconcurrentusers="name(user1)host(host1)"`
    `-qexemptconcurrentusers="name(user2)host(host1)"`
  - `-qexemptconcurrentusers="name(user1)host(host1) | name(user2)host(host1)"`

  For _user_name_, _user_ID_, and _host_name_, you can also use a list of user names, user IDs, or hostnames separated by a space within the parentheses. For example:

```
-qexemptconcurrentusers="name(user1 user2)host(host1)"
```

This is equivalent to the previous examples.

Note: Specifying this entry does not exempt users from compiler utilization tracking. It only exempts them from being counted as concurrent users. To optimize utilization tracking performance, the format of the specified value is not validated until the report is produced. For more information about counting concurrent users, see "Concurrent user considerations" on page 49.

-qqualhostname | **-qnoqualhostname**

Specifies whether host names that are captured in usage files and then listed in compiler usage reports are qualified with domain names.

If all compiler usage within your organization is on machines within a single domain, you can reduce the size of the usage files by using **-qnoqualhostname** to suppress domain name qualification.

Default: **-qqualhostname**, which means the host names are qualified with domain names.

-qenabletracking | **-qnoenabletracking**

Enables or disables utilization tracking.

Default: **-qnoenabletracking**, which means utilization tracking is disabled.

-qusagefileloc=*directory_or_ file_name*

Specifies the location of the usage file.

By default, a `.cuf` file is automatically created for each user in their home directory. You can set up a central location where the files for each user can be created. For more information, see "Usage file location" on page 50.

The following rules apply when you specify this entry:
- If a file name is specified, it must have the `.cuf` extension. If the file is a symlink, it must point to a file with the `.cuf` extension. If the specified file does not exist, a `.cuf` file is created, along with any parent directories that do not already exist.
- If a directory is specified, there must be exactly one file with the `.cuf` extension in the directory. A new file is not created in this case.
- The path of the specified directory can be a relative or an absolute path. Relative paths are relative to the compiler user's current working directory.

Note: If a compiler user cannot access the file, for example, because of insufficient permissions to use or create the file, the compiler generates a message and the compilation continues.

You can use the following variables for this option:
- *$HOME* for the user's home directory. This allows each user to have a `.cuf` file in their home directory or a subdirectory of their home directory.
- *$USER* or *$LOGNAME* for the user's login user name. You can use this variable to create a `.cuf` file for each user and include the user's login name in the name of the `.cuf` file or in the name of a parent directory.
- *$HOSTNAME* for the name of the host on which the compiler runs. This can be useful when you track utilization across different hosts.

-qfileaccessmaxwait=*number_of_milliseconds*

Specifies the maximum number of milliseconds to wait for accessing the usage file.

Note: This entry is used to account for unusual circumstances where the system is under extreme heavy load and there is a delay in accessing the usage file.

Default: 3000 milliseconds

Notes:
- These entries are not compiler options. They can only be used in the utilization tracking configuration file.
- If the **-qexemptconcurrentusers** entry is specified multiple times in the utilization tracking configuration file, all the specified instances are used. If other entries are specified multiple times, the value of the last one overrides previous ones.

Understanding the utilization reporting tool

You can use the utilization reporting tool to generate compiler usage reports from the information in one or more usage files, and optionally prune the usage files when you generate the reports.

The tool is located in the `/opt/ibmurt/1.1/bin` directory. You can use the **urt** command to invoke it. The syntax of the **urt** command is as follows:
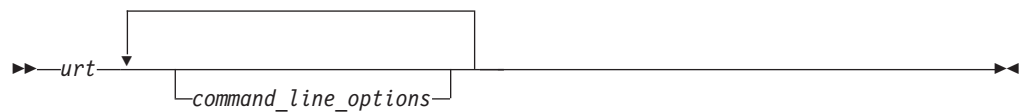
```
►►─urt─┬──────────────────────┬─────────────────────►◄
       └─command_line_options─┘
```

The generated report is displayed on the standard output. You can direct the output to a file if you want to keep the report.

Command-line options control how usage reports are generated. For more information about the options, see "Utilization reporting tool command-line options" on page 58.

A default configuration file `ibmurt.cfg` is provided in the `/opt/ibmurt/1.1/config` directory. Entries in this file take the same form as the command-line options and have the same effect. You can also create additional configuration files and use the **-qconfigfile** option to specify their names.

You can specify the options in one or more of the following places:
1. The default configuration file
2. The additional configuration file specified with **-qconfigfile**
3. The command line

The utilization reporting tool uses the options in the default configuration file before it uses the options on the command line. When it encounters a **-qconfigfile** option on the command line, it reads the options in the specified configuration file and puts them on the command line at the place where the **-qconfigfile** option is used.

If an option is specified multiple times, the last specification that the tool encounters takes effect. Exceptions are **-qconfigfile** and **-qsameuser**. For these two options, all specifications take effect.

Utilization reporting tool command-line options

The utilization reporting tool command-line options control the generation of the compiler utilization report.

Use these command-line options to modify the details of your compiler utilization report.

-qreporttype=detail | maxconcurrent

Specifies the type of the usage report to generate.

- **detail** specifies that all invocations of the compiler are listed in the usage report. With this suboption, you can get a detailed report, in which the invocations that have exceeded the allowed maximum number of concurrent users are indicated.
- **maxconcurrent** specifies that only the compiler invocations that have exceeded the allowed maximum number of concurrent users are listed. With this suboption, you can get a compact report, which does not list those invocations within the maximum number of allowed concurrent users.

Note: The allowed maximum number of concurrent users is specified with the **-qmaxconcurrentusers** entry in the utilization tracking configuration file.

Default: **-qreporttype=maxconcurrent**.

-qrptmaxrecords=_num_ **| nomax**

Specifies the maximum number of records to list in the report for each product. _num_ must be a positive integer.

Default: **-qrptmaxrecords=nomax**, which means all the records are listed.

-qusagefileloc=_directory_or_file_name_

Specifies the location of the usage files for report generation or pruning. It can be a list of directories or file names, or both, separated by colons.

The following rules apply when you specify this option:

- If one or more directories are specified, all files with the `.cuf` extension in those directories are used. Subdirectories can also be searched by using the **-qmaxsubdirs** option.
- The path of the specified directory can be relative or absolute. Relative paths are relative to the compiler user's current working directory.
- A symlink does not require the `.cuf` extension but the file to which it points must have that extension.

Note:

- The **-qusagefileloc** entry in the utilization tracking configuration file tells the compiler which usage files to use for recording compiler utilization. This **-qusagefileloc** option tells the utilization reporting tool where to find those usage files.

Default: _.:$HOME_, which means the utilization reporting tool looks for usage files in your current working directory and your home directory.

-qmaxsubdirs=_num_ **| nomax**

Specifies whether to search subdirectories for usage files, and how many levels of subdirectories to search. *num* must be a non-negative integer.

If **nomax** is specified, all the subdirectories are searched. If 0 is specified, no subdirectories are searched.

Default: 0.

-qconfigfile=_file_path_

Specifies the user defined configuration file that you want to use.

For more information about how the utilization reporting tool uses the configuration file, see "Understanding the utilization reporting tool" on page 57.

Note: If you specify this option multiple times, all specified instances are used.

-qsameuser=_user_account_info_

Specifies different user accounts that belong to the same compiler user. Use this option when a user accesses the compiler from more than one user ID or machine to avoid having that user reported as multiple users. Invocations of the compiler by these different accounts are counted as a single user instead of multiple different users.

user_account_info can be any combination of the following items:

- name(*user_name*)
- uid(*user_ID*)
- host(*host_name*)

There are two ways to pass these rules to the utilization reporting tool. You can supply specific lists of the *user_name*s, *user_ID*s or*host_name*s that are shared by the same user or you can use a more generic (=) syntax.

For example, to indicate that *user1* and *user2* are both user names belonging to the same person who uses the compiler on the *host1* machine, use the syntax in which you specify these user names and the host name explicitly:

```
-qsameuser="name(user1)host(host1) | name(user2)host(host1)"
```

or

```
-qsameuser="name(user1 user2)host(host1)"
```

Both of these examples use specific user names and host names to indicate accounts that belong to the same user, but they do so in slightly different ways. The first example uses a vertical bar to separate the different user accounts that belong to this user, while the second example uses a list of user names within the parentheses instead of repeating the same host information twice. They both convey the same account information, but the second example is more concise.

As an example of the more generic (=) syntax, you can indicate that all user accounts with the same user name and uid belong to the same user as follows:

```
-qsameuser="name(=)uid(=)"
```

With this option, you are not specifying specific user names or uids as you did in the previous example. User accounts that have the same user name and uid are considered as belonging to the same user, regardless of what the specific user names and uids are, and regardless of what the host name is. This establishes a general rule that applies to all accounts in your organization instead of specific ones.

The following rules apply when you specify this option:
- Each instance of the **-qsameuser** option must use either the list or generic (=) syntax. You cannot combine them in the same instance of the option but you can use multiple instances of the **-qsameuser** option to refine the report.
- The utilization reporting tool matches the user information based on the order that the **-qsameuser** option values are specified. Once it finds a match it stops matching the same user information against any subsequent options.

  The following examples illustrates the differences:
  - If you specify the **-qsameuser** option as follows:

    ```
    -qsameuser="name(user1)" -qsameuser="uid(=)"
    ```

    Specifying the **-qsameuser** option in this order means that user accounts with the user name *user1* matches the first option and is not evaluated against the second option. User accounts *user1* and *user2* are not considered the same user even if they have the same *uid*.
  - If you specify the **-qsameuser** option as follows:

    ```
    -qsameuser="uid(=)" -qsameuser="name(user1)"
    ```

    Specifying the **-qsameuser** option in this order means that user accounts with the same *uid* are always considered to be the same user, and in addition, any user accounts with a user name of *user1* should be considered belonging to the same user even if they do not match by *uid*.

Note: Specifying this option does not prevent user information from being listed in the usage report. For more information about concurrent users, see "Concurrent user considerations" on page 49.

-qadjusttime=*time_adjustments*

Adjusts the time that have been recorded in the usage files for the specified machines. *time_adjustments* is a list of entries with the format of *machine name + | - number of seconds*, separated by colons.

The following rules apply when you use this option:
- An entry of the form *machine name + number of seconds* causes the specified number of seconds to be added to the start and end times of any invocations recorded for the specified machine.
- An entry of the form *machine name - number of seconds* causes the specified number of seconds to be subtracted from the start and end times of any invocations recorded for the specified machine.

For example:

```
-qadjusttime="hostA+5:hostB-3"
```

Five seconds are added to the start and end times of the invocations on `hostA`, and three seconds are subtracted from the start and end times of the invocations on `hostB`.

Only use this option if the usage files contain utilization information from two or more machines, and time is not synchronized across those machines. The adjustments specified by this option compensate for the lack of synchronization

Notes:

- The specified adjustments are only used for the current run of the **urt** command. Specifying this option does not change the invocation information recorded in the usage files.
- Do not specify the same machine name more than once with this option.

-qusagefilemaxage=*number_of_days* **|** **nomax**

Prunes the usage files by removing all invocations older than the specified number of days.

Every usage file specified by the **-qusagefileloc** option is pruned. The usage report contains this information to indicate the number of records that have been pruned.

Default: **-qusagefilemaxage=nomax**, which means no pruning is performed.

-qusagefilemaxsize=*number_of_MB* **|** **nomax**

Prunes the usage files to keep them under the specified size. It prunes the files by removing the oldest invocations.

Every usage file specified by the **-qusagefileloc** option is pruned. The usage report contains this information to indicate the number of records that have been pruned.

Default: **-qusagefilemaxsize=nomax**, which means no pruning is performed.

-qtimesort=ascend **|** **descend**

Specifies the chronological order in which the usage report information is sorted.
- Specifying **ascend** means new information is listed after the older information.
- Specifying **descend** means the newest information is at the top of the report.

Default: **-qtimesort=ascend**.

Generating usage reports

You can use the utilization reporting tool to generate compiler usage reports based on the usage information stored in the usage files.

To generate a report, use the command-line options or the **urt** configuration file to specify how you want a report to be generated. For more information about these options, see "Utilization reporting tool command-line options" on page 58.

Notes:
- You can set up a cron service to run the utilization reporting tool on a regular basis. If the usage files from which the tool generate reports need to be copied to the machine where the tool is running, you can also automate this copying task with cron.
- To reduce contention for read and write access to the usage files, do not run the tool or copy the usage files when the compiler is being used.

The generated report is displayed on the standard output. You can direct the output to a file if you want to keep the report.

After a usage report is generated, the utilization reporting tool uses the following exit codes to indicate the compliance status of your compiler license:
- Exit code ="1".

The utilization reporting tool has detected that the number of Concurrent User license entitlements specified in the **-qmaxconcurrentusers** entry in the utilization tracking configuration file has been exceeded. See the generated report for details and contact your IBM representative to purchase additional Concurrent User licenses.

- Exit code ="0".

  The compiler utilization is within the number of Concurrent User license entitlements specified.

For more information about the **urt** command, see "Understanding the utilization reporting tool" on page 57.

Understanding usage reports

You can use the report that the utilization report tool generates to analyze the compiler usage in your organization.

The report has a REPORT SUMMARY section that lists the following information:

1. The date and time when the report is generated.
2. The `.cuf` file or a list of all `.cuf` files used to generate the report.
3. The options that have been passed to the **urt** command, with default values for any unspecified options.
4. Possible messages categorized as ERROR, WARNING, or INFO. For detailed information about possible messages, see "Diagnostic messages from utilization tracking and reporting" on page 65.

After the summary section, there is a REPORT DETAILS section for each compiler version. This section lists all of the compiler invocations recorded in the usage files. The content of these sections varies depending on the report type that you have specified. For detailed information about the report types, see **-qreporttype**.

Here are the sample reports generated with the two different report types:

Sample 1: A sample report generated with **-qreporttype=detail**

```
REPORT SUMMARY
--------------

DATE: 12/18/09     TIME: 01:30:24

OPTIONS USED (* indicates that a default value was used):

reporttype=detail
maxsubdirs=0
configfile="/opt/ibmurt/1.1/config/ibmurt.cfg"
rptmaxrecords=nomax
*adjusttime=
usagefileloc="/home/testrun/ibmxlcompiler.cuf"
*sameuser=
timesort=ascend
usagefilemaxsize=nomax
usagefilemaxage=nomax


FILES USED:

/home/testrun/ibmxlcompiler.cuf

REPORT DETAILS
--------------
```

```
USAGE INFORMATION FOR PRODUCT: IBM XL C/C++ for Linux 11.1

Max. Concurrent Users Exceeded? : *** YES ***

Max. Concurrent Users Allowed: 1          Max. Concurrent Users Recorded: 5

Exempt Users:

Product invocations:

Start Time          End Time           User              Number of Concurrent Users
------------------  ------------------ -----------------  --------------------------
12/17/09 16:56:44   12/17/09 16:57:13  user1@host1.ibm.com 1
12/18/09 00:58:29   12/18/09 00:58:32  user2@host2.ibm.com 1
12/18/09 01:16:01   12/18/09 01:16:02  user3@host3.ibm.com 5 ( exceeds max. allowed)
12/18/09 01:16:02   12/18/09 01:16:26  user2@host2.ibm.com 5 ( exceeds max. allowed)
12/18/09 01:16:08   12/18/09 01:16:08  user3@host2.ibm.com 5 ( exceeds max. allowed)
12/18/09 01:16:12   12/18/09 01:16:12  user2@host1.ibm.com 5 ( exceeds max. allowed)
12/18/09 01:16:24   12/18/09 01:16:28  user1@host2.ibm.com 5 ( exceeds max. allowed)
12/18/09 01:26:11   12/18/09 01:27:46  user3@host3.ibm.com 2 ( exceeds max. allowed)
12/18/09 01:26:27   12/18/09 01:27:46  user1@host1.ibm.com 2 ( exceeds max. allowed)
12/18/09 01:29:59   12/18/09 01:30:00  user2@host1.ibm.com 1
12/18/09 01:30:00   12/18/09 01:30:00  user2@host2.ibm.com 3 ( exceeds max. allowed)
12/18/09 01:30:14   12/18/09 01:30:15  user3@host1.ibm.com 3 ( exceeds max. allowed)
12/18/09 01:30:14   12/18/09 01:30:14  user2@host2.ibm.com 3 ( exceeds max. allowed)
```

Sample 2: A sample report generated with **-qreporttype=maxconcurrent**

```
REPORT SUMMARY
--------------

DATE: 12/18/09      TIME: 01:32:53

OPTIONS USED (* indicates that a default value was used):

reporttype=maxconcurrent
maxsubdirs=0
configfile="/opt/ibmurt/1.1/config/ibmurt.cfg"
rptmaxrecords=nomax
*adjusttime=
usagefileloc="/home/testrun/ibmxlcompiler.cuf"
*sameuser=
timesort=ascend
usagefilemaxsize=nomax
usagefilemaxage=nomax


FILES USED:

/home/testrun/ibmxlcompiler.cuf

REPORT DETAILS
--------------

USAGE INFORMATION FOR PRODUCT: IBM XL C/C++ for Linux 11.1

Max. Concurrent Users Exceeded? : *** YES ***

Max. Concurrent Users Allowed: 1          Max. Concurrent Users Recorded: 5

Exempt Users:

Dates and times where usage exceeded the maximum allowed:

Date          Time          Number of Concurrent Users      Users
------------  ----          --------------------------      -----
```

12/18/09	01:16:01	5	user3@host3.ibm.com
			user2@host2.ibm.com
			user3@host2.ibm.com
			user2@host1.ibm.com
			user1@host2.ibm.com
12/18/09	01:16:02	5	user3@host3.ibm.com
			user2@host2.ibm.com
			user3@host2.ibm.com
			user2@host1.ibm.com
			user1@host2.ibm.com
12/18/09	01:16:08	5	user3@host3.ibm.com
			user2@host2.ibm.com
			user3@host2.ibm.com
			user2@host1.ibm.com
			user1@host2.ibm.com
12/18/09	01:16:12	5	user3@host3.ibm.com
			user2@host2.ibm.com
			user3@host2.ibm.com
			user2@host1.ibm.com
			user1@host2.ibm.com
12/18/09	01:16:24	5	user3@host3.ibm.com
			user2@host2.ibm.com
			user3@host2.ibm.com
			user2@host1.ibm.com
			user1@host2.ibm.com
12/18/09	01:26:11	2	user3@host3.ibm.com
			user1@host1.ibm.com
12/18/09	01:26:27	2	user3@host3.ibm.com
			user1@host1.ibm.com
12/18/09	01:30:00	3	user2@host2.ibm.com
			user2@host1.ibm.com
			user3@host1.ibm.com
12/18/09	01:30:14	3	user2@host2.ibm.com
			user2@host1.ibm.com
			user3@host1.ibm.com
12/18/09	01:30:14	3	user2@host2.ibm.com
			user2@host1.ibm.com
			user3@host1.ibm.com

Note: There are circumstances under which an end time might not be recorded. These might include:

- There was a major failure of the compiler, for example, power loss during a compilation.
- The invocation had not ended at the time when the report was generated, or at the time when the usage file was being copied.
- The permission to write to the usage file was revoked at some time before the end time of the invocation was recorded.

An invocation with no end time recorded is not included in the count of concurrent users.

Pruning usage files

Usage files grow with each compiler invocation. You can prune the usage files with the utilization report tool.

When you generate a usage report, you can specify the following two options to optionally prune the usage files:

- **-qusagefilemaxage**: Removes the invocations older than the specified number of days. For example, to remove all entries in the usage files older than 30 days, use the following command:

```
urt -qusagefilemaxage=30
```

- **-qusagefilemaxsize**: Removes the oldest invocations to keep the usage files under the specified size. For example, to remove the oldest invocations to keep the usage files under 30 MB, use the following command:

```
urt -qusagefilemaxsize=30
```

When usage files are pruned, the usage report includes an information message that indicates the number of records that have been pruned. If you want to keep the generated report after the files are pruned, you can redirect the output to a file.

To control the size of the usage files, you can prune the usage files on a regular basis. You can create a cron job to do this automatically.

If you do not have the utilization reporting tool installed on each machine where the usage files are located, you have the following options:

- Export the file system from each machine where the usage files exist and mount it on the machine where the utilization reporting tool is installed. Then run the tool to prune the usage files on the mounted network file system.
- After copying the usage files to the machine where the utilization reporting tool is installed, delete the files and use new usage files to capture any subsequent compiler invocations. This approach might also speed up the report generation because the utilization reporting tool is not accessing the usage files remotely and it is not spending time pruning the usage files.

Pruning usage files might slow down the usage report generation process, especially when the number or the size of the usage files is large. If you do not want to prune the files every time you generate reports, you can set the values for the **-qusagefilemaxage** and **-qusagefilemaxsize** options as follows:

- If you generate the report daily, you can specify these two options with very high values so pruning does not occur. The default value **nomax** can be used in this case.
- You can set appropriate values for these two options and use a separate cron job to prune the usage files weekly.

Note: To reduce contention for read and write access to the usage files, do not run the utilization report tool or copy the usage files when the compiler is being used.

Diagnostic messages from utilization tracking and reporting

The compiler generates diagnostic messages to indicate utilization tracking issues. These messages can help you to fix the associated problems.

For example:

```
Utilization tracking configuration file could not be read due to
insufficient permissions.
```

This message indicates that you need read access for utilization tracking configuration file.

When the utilization reporting tool is used to generate usage reports or prune usage files, it also generates diagnostic messages. For example:

```
Unrecognized option -qmaxsubdir.
```

This message indicates that you have specified a wrong option.

Note: Possible error, warning, or information messages are also included in the compiler usage report generated by the tool.

Chapter 4. Compiler options reference

The following sections contain a summary of the compiler options available in XL C/C++ by functional category, followed by detailed descriptions of the individual options.

Related information
- "Specifying compiler options" on page 5
- "Reusing GNU C/C++ compiler options with **gxlc** and **gxlc++**" on page 10

Summary of compiler options by functional category

The XL C/C++ options available on the Linux platform are grouped into the following categories. If the option supports an equivalent pragma directive, this is indicated. To get detailed information on any option listed, see the full description for that option.
- "Output control"
- "Input control" on page 68
- "Language element control" on page 69
- "Template control (C++ only)" on page 70
- "Floating-point and integer control" on page 71
- "Error checking and debugging" on page 74
- "Listings, messages, and compiler information" on page 76
- "Optimization and tuning" on page 78
- "Object code control" on page 72
- "Linking" on page 81
- "Portability and migration" on page 82
- "Compiler customization" on page 83

Output control

The options in this category control the type of file output the compiler produces, as well as the locations of the output. These are the basic options that determine the compiler components that will be invoked; the preprocessing, compilation, and linking steps that will (or will not) be taken; and the kind of output to be generated.

Table 9. Compiler output options

Option name	Equivalent pragma name	Description
"-c" on page 105	None.	Prevents the completed object from being sent to the linker. With this option, the output is a .o file for each source file.
"-C, -C!" on page 106	None.	When used in conjunction with the **-E** or **-P** options, preserves or removes comments in preprocessed output.
"-E" on page 127	None.	Preprocesses the source files named in the compiler invocation, without compiling, and writes the output to the standard output.

Table 9. Compiler output options (continued)

Option name	Equivalent pragma name	Description
"-qmakedep, -M" on page 217	None.	Creates an output file containing targets suitable for inclusion in a description file for the **make** command.
"-MF" on page 223	None.	Specifies the target for the output generated by the **-qmakedep** or **-M** options.
"-qmkshrobj" on page 224	None.	Creates a shared object from generated object files.
"-o" on page 226	None.	Specifies a name for the output object, assembler, or executable file.
"-P" on page 232	None.	Preprocesses the source files named in the compiler invocation, without compiling, and creates an output preprocessed file for each input file.
"-S" on page 262	None.	Generates an assembler language file for each source file.
"-qshowmacros" on page 267	None.	Emits macro definitions to preprocessed output.
"-qtimestamps" on page 303	None.	Controls whether or not implicit time stamps are inserted into an object file.

Input control

The options in this category specify the type and location of your source files.

Table 10. Compiler input options

Option name	Equivalent pragma name	Description
"-+ (plus sign) (C++ only)" on page 85	None.	Compiles any file as a C++ language file.
"-qcinc (C++ only)" on page 112	None.	Places an `extern "C" { }` wrapper around the contents of include files located in a specified directory.
"-I" on page 158	None.	Adds a directory to the search path for include files.
"-qidirfirst" on page 159	#pragma options idirfirst	Specifies whether the compiler searches for user include files in directories specified by the **-I** option *before* or *after* searching any other directories.
"-qinclude" on page 162	None.	Specifies additional header files to be included in a compilation unit, as though the files were named in an `#include` statement in the source file.

Table 10. Compiler input options (continued)

Option name	Equivalent pragma name	Description
"-qsourcetype" on page 277	None.	Instructs the compiler to treat all recognized source files as a specified source type, regardless of the actual file name suffix.
"-qstdinc" on page 285	#pragma options stdinc	Specifies whether the standard include directories are included in the search paths for system and user header files.

Language element control

The options in this category allow you to specify the characteristics of the source code. You can also use these options to enforce or relax language restrictions, and enable or disable language extensions.

Table 11. Language element control options

Option name	Equivalent pragma name	Description
"-qaltivec" on page 94	None	Enables compiler support for vector data types and operators.
"-qasm" on page 98	None	Controls the interpretation of and subsequent generation of code for assembler language extensions.
"-qcpluscmt (C only)" on page 116	None.	Enables recognition of C++-style comments in C source files.
"-D" on page 120	None.	Defines a macro as in a #define preprocessor directive.
"-qdigraph" on page 124	#pragma options digraph	Enables recognition of digraph key combinations or keywords to represent characters not found on some keyboards.
"-qdollar" on page 125	#pragma options dollar	Allows the dollar-sign ($) symbol to be used in the names of identifiers.
"-qignprag" on page 161	#pragma options ignprag	Instructs the compiler to ignore certain pragma statements.
"-qkeyword" on page 185	None.	Controls whether the specified name is treated as a keyword or as an identifier whenever it appears in your program source.
"-qlanglvl" on page 188	▶ C #pragma options langlvl, #pragma langlvl	Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.

Table 11. Language element control options (continued)

Option name	Equivalent pragma name	Description
"-qlonglong" on page 216	#pragma options long long	Allows IBM `long long` integer types in your program.
"-qmbcs, -qdbcs" on page 221	#pragma options mbcs, #pragma options dbcs	Enables support for multibyte character sets (MBCS) and Unicode characters in your source code.
"-qstaticinline (C++ only)" on page 282	None.	Controls whether inline functions are treated as having `static` or `extern` linkage.
"-qtabsize" on page 295	None.	Sets the default tab length, for the purposes of reporting the column number in error messages.
"-qtrigraph" on page 307	None.	Enables the recognition of trigraph key combinations to represent characters not found on some keyboards.
"-U" on page 311	None.	Undefines a macro defined by the compiler or by the **-D** compiler option.
"-qutf" on page 316	None.	Enables recognition of UTF literal syntax.

Template control (C++ only)

You can use these options to control how the C++ compiler handles templates.

Table 12. C++ template options

Option name	Equivalent pragma name	Description
"-qtempinc (C++ only)" on page 297	None.	Generates separate template include files for template functions and class declarations, and places these files in a directory which can be optionally specified.
"-qtemplatedepth (C++ only)" on page 298	None.	Specifies the maximum number of recursively instantiated template specializations that will be processed by the compiler.
"-qtemplaterecompile (C++ only)" on page 299	None.	Helps manage dependencies between compilation units that have been compiled using the **-qtemplateregistry** compiler option.

Option name	Equivalent pragma name	Description
"-qtemplateregistry (C++ only)" on page 300	None.	Maintains records of all templates as they are encountered in the source and ensures that only one instantiation of each template is made.
"-qtempmax (C++ only)" on page 302	None.	Specifies the maximum number of template include files to be generated by the **-qtempinc** option for each header file.
"-qtmplinst (C++ only)" on page 305	None.	Manages the implicit instantiation of templates.
"-qtmplparse (C++ only)" on page 306	None.	Controls whether parsing and semantic checking are applied to template definitions.
▶ C++0x -qlanglvl=[no]externtemplate	None.	Suppresses the implicit instantiation of a template specialization or its members.
-qlanglvl=[no]gnu_externtemplate	None.	Suppresses the implicit instantiation of a template specialization or its members. This option is deprecated in XL C/C++ V11.1; you can use the option **-qlanglvl=[no]externtemplate** instead.
▶ C++0x -qlanglvl=[no]variadic[templates]	None.	Defines class or function templates that can have any number (including zero) of parameters.

Floating-point and integer control

Specifying the details of how your applications perform calculations can allow you to take better advantage of your system's floating-point performance and precision, including how to direct rounding. However, keep in mind that strictly adhering to IEEE floating-point specifications can impact the performance of your application. Using the options in the following table, you can control trade-offs between floating-point performance and adherence to IEEE standards.

Table 13. Floating-point and integer control options

Option name	Equivalent pragma name	Description
"-qbitfields" on page 104	None.	Specifies whether bit fields are signed or unsigned.

Table 13. Floating-point and integer control options (continued)

Option name	Equivalent pragma name	Description
"-qchars" on page 109	#pragma options chars, #pragma chars	Determines whether all variables of type char are treated as either signed or unsigned.
"-qenum" on page 129	#pragma options enum, #pragma enum	Specifies the amount of storage occupied by enumerations.
"-qfloat" on page 137	#pragma options float	Selects different strategies for speeding up or improving the accuracy of floating-point calculations.
"-qldbl128" on page 206	#pragma options ldbl128	Increases the size of long double types from 64 bits to 128 bits.
"-qlonglit" on page 215	None.	In 64-bit mode, when determining the implicit types for integer literals, the compiler behaves as if an l or L suffix were added to integral literals with no suffix or with a suffix consisting only of u or U.
"-y" on page 327	None.	Specifies the rounding mode for the compiler to use when evaluating constant floating-point expressions at compile time.

Object code control

These options affect the characteristics of the object code, preprocessed code, or other output generated by the compiler.

Table 14. Object code control options

Option name	Equivalent pragma name	Description
"-q32, -q64" on page 87	None.	Selects either 32-bit or 64-bit compiler mode.
"-qalloca, -ma (C only)" on page 93	#pragma alloca	Provides an inline definition of system function alloca when it is called from source code that does not include the alloca.h header.
"-qcommon" on page 113	None.	Controls where uninitialized global variables are allocated.
"-qeh (C++ only)" on page 129	None.	Controls whether exception handling is enabled in the module being compiled.
"-qinlglue" on page 172	#pragma options inlglue	When used with **-O2** or higher optimization, inlines glue code that optimizes external function calls in your application.
"-qpic" on page 243	None.	Generates Position-Independent Code suitable for use in shared libraries.

Table 14. Object code control options (continued)

Option name	Equivalent pragma name	Description
"-qppline" on page 244	None.	When used in conjunction with the **-E** or **-P** options, enables or disables the generation of #line directives.
"-qpriority (C++ only)" on page 249	#pragma options priority, #pragma priority	Specifies the priority level for the initialization of static objects.
"-qproto (C only)" on page 252	#pragma options proto	Specifies the linkage conventions for passing floating-point arguments to functions that have not been prototyped.
"-r" on page 253	None.	Produces a nonexecutable output file to use as an input file in another ld command call. This file may also contain unresolved symbols.
"-qreserved_reg" on page 256	None.	Indicates that the given list of registers cannot be used during the compilation except as a stack pointer, frame pointer or in some other fixed role.
"-qro" on page 258	#pragma options ro, #pragma strings	Specifies the storage type for string literals.
"-qroconst" on page 260	#pragma options roconst	Specifies the storage location for constant values.
"-qrtti (C++ only)" on page 261	None.	Generates runtime type identification (RTTI) information for exception handling and for use by the typeid and dynamic_cast operators.
"-s" on page 262	None.	Strips the symbol table, line number information, and relocation information from the output file.
"-qsaveopt" on page 263	None.	Saves the command-line options used for compiling a source file, the user's configuration file name and the options specified in the configuration files, the version and level of each compiler component invoked during compilation, and other information to the corresponding object file.

Table 14. Object code control options (continued)

Option name	Equivalent pragma name	Description
"-qstackprotect" on page 281	None.	Provides protection against malicious code or programming errors that overwrite or corrupt the stack.
"-qstatsym" on page 284	None.	Adds user-defined, nonexternal names that have a persistent storage class, such as initialized and uninitialized static variables, to the symbol table of the object file.
"-qtbtable" on page 296	#pragma options tbtable	Controls the amount of debugging traceback information that is included in the object files.
"-qthreaded" on page 302	None.	Indicates to the compiler whether it must generate threadsafe code.
"-qtls" on page 304	None.	Enables recognition of the __thread storage class specifier, which designates variables that are to be allocated threadlocal storage; and specifies the threadlocal storage model to be used.
"-qvrsave" on page 319	#pragma altivec_vrsave	Enables code in function prologs and epilogs to maintain the VRSAVE register.
"-qxcall" on page 325	None.	Generates code to treat static functions within a compilation unit as if they were external functions.

Error checking and debugging

The options in this category allow you to detect and correct problems in your source code. In some cases, these options can alter your object code, increase your compile time, or introduce runtime checking that can slow down the execution of your application. The option descriptions indicate how extra checking can impact performance.

To control the amount and type of information you receive regarding the behavior and performance of your application, consult the options in "Listings, messages, and compiler information" on page 76.

For information on debugging optimized code, see the *XL C/C++ Optimization and Programming Guide*.

Table 15. Error checking and debugging options

Option name	Equivalent pragma name	Description
"-# (pound sign)" on page 86	None.	Previews the compilation steps specified on the command line, without actually invoking any compiler components.
"-qcheck" on page 110	#pragma options check	Generates code that performs certain types of runtime checking.
"-qflttrap" on page 141	#pragma options flttrap	Determines the types of floating-point exception conditions to be detected at run time
"-qformat" on page 144	None.	Warns of possible problems with string input and output format specifications.
"-qfullpath" on page 145	#pragma options fullpath	When used with the **-g** or **-qlinedebug** option, this option records the full, or absolute, path names of source and include files in object files compiled with debugging information, so that debugging tools can correctly locate the source files.
"-qfunctrace" on page 146	None.	Calls to the tracing routine that traces the entry and exit points of functions in a compilation unit, or only a specific list of functions.
"-g" on page 149	None.	Generates debug information for use by a symbolic debugger.
"-qhalt" on page 153	#pragma options halt	Stops compilation before producing any object, executable, or assembler source files if the maximum severity of compile-time messages equals or exceeds the severity you specify.
"-qhaltonmsg (C++ only)" on page 154	None.	Stops compilation before producing any object, executable, or assembler source files if a specified error message is generated.
"-qinfo" on page 164	#pragma options info, #pragma info	Produces or suppresses groups of informational messages.
"-qinitauto" on page 171	#pragma options initauto	Initializes uninitialized automatic variables to a specific value, for debugging purposes.
"-qkeepparm" on page 184	None.	When used with **-O2** or higher optimization, specifies whether function parameters are stored on the stack.

Table 15. Error checking and debugging options (continued)

Option name	Equivalent pragma name	Description
"-qlinedebug" on page 209	None.	Generates only line number and source file name information for a debugger.
"-qmaxerr" on page 219	None.	Halts compilation when a specified number of errors of a specified severity level or higher is reached.
"-qoptdebug" on page 230	None.	When used with high levels of optimization, produces files containing optimized pseudocode that can be read by a debugger.
"-qsymtab (C only)" on page 292	None.	Determines the information that appears in the symbol table.
"-qsyntaxonly (C only)" on page 293	None.	Performs syntax checking without generating an object file.
"-qwarn0x (C++0x)" on page 323	None.	Controls whether to inform users with messages about differences in their programs caused by migration from the C++98 standard to the C++0x standard.
"-qwarn64" on page 324	None.	Enables checking for possible data conversion problems between 32-bit and 64-bit compiler modes.

Listings, messages, and compiler information

The options in this category allow you control over the listing file, as well as how and when to display compiler messages. You can use these options in conjunction with those described in "Error checking and debugging" on page 74 to provide a more robust overview of your application when checking for errors and unexpected behavior.

Table 16. Listings and messages options

Option name	Equivalent pragma name	Description
"-qattr" on page 102	#pragma options attr	Produces a compiler listing that includes the attribute component of the attribute and cross-reference section of the listing.
"-qdump_class_hierarchy (C++ only)" on page 126	None.	Dumps a representation of the hierarchy and virtual function table layout of each class object to a file.
"-qflag" on page 136	#pragma options flag, ▶ C++ "#pragma report (C++ only)" on page 368	Limits the diagnostic messages to those of a specified severity level or higher.

Table 16. Listings and messages options (continued)

Option name	Equivalent pragma name	Description
"-qlist" on page 210	#pragma options list	Produces a compiler listing file that includes an object listing.
"-qlistfmt" on page 212	None.	Creates an XML report to assist with finding optimization opportunities.
"-qlistopt" on page 214	None.	Produces a compiler listing file that includes all options in effect at the time of compiler invocation.
"-qphsinfo" on page 242	None.	Reports the time taken in each compilation phase to standard output.
"-qprint" on page 248	None.	Enables or suppresses listings.
"-qreport" on page 254	None.	Produces listing files that show how sections of code have been optimized.
"-qshowinc" on page 266	#pragma options showinc	When used with **-qsource** option to generate a listing file, selectively shows user or system header files in the source section of the listing file.
"-qskipsrc" on page 270	None.	When a listing file is generated using the **-qsource** option, **-qskipsrc** can be used to determine whether the source statements skipped by the compiler are shown in the source section of the listing file. Alternatively, the **-qskipsrc=hide** option is used to hide the source statements skipped by the compiler.
"-qsource" on page 276	#pragma options source	Produces a compiler listing file that includes the source section of the listing and provides additional source information when printing error messages.
"-qsrcmsg (C only)" on page 280	None.	Adds the corresponding source code lines to diagnostic messages generated by the compiler.
"-qsuppress" on page 291	None.	Prevents specific informational or warning messages from being displayed or added to the listing file, if one is generated.

Table 16. Listings and messages options (continued)

Option name	Equivalent pragma name	Description
"-v, -V" on page 316	None.	Reports the progress of compilation, by naming the programs being invoked and the options being specified to each program.
"-qversion" on page 317	None.	Displays the version and release of the compiler being invoked.
"-w" on page 320	None.	Suppresses informational, language-level and warning messages.
"-qxref" on page 325	#pragma options xref	Produces a compiler listing that includes the cross-reference component of the attribute and cross-reference section of the listing.

Optimization and tuning

The options in this category allow you to control the optimization and tuning process, which can improve the performance of your application at run time.

Remember that not all options benefit all applications. Trade-offs sometimes occur between an increase in compile time, a reduction in debugging capability, and the improvements that optimization can provide.

You can also control some of these options, such as **Optimize**, **-qcompact**, or **-qstrict**, with an **option_override** pragma.

In addition to the option descriptions in this section, consult the *XL C/C++ Optimization and Programming Guide* for a details on the optimization and tuning process as well as writing optimization-friendly source code.

Table 17. Optimization and tuning options

Option name	Equivalent pragma name	Description
"-qaggrcopy" on page 88	None.	Enables destructive copy operations for structures and unions.
"-qalias" on page 89	None.	Indicates whether a program contains certain categories of aliasing or does not conform to C/C++ standard aliasing rules. The compiler limits the scope of some optimizations when there is a possibility that different names are aliases for the same storage location..
"-qarch" on page 95	None.	Specifies the processor architecture for which the code (instructions) should be generated.

Table 17. Optimization and tuning options (continued)

Option name	Equivalent pragma name	Description
"-qcache" on page 106	None.	When specified with **-O4**, **-O5**, or **-qipa**, specifies the cache configuration for a specific execution machine.
"-qcompact" on page 114	#pragma options compact	Avoids optimizations that increase code size.
"-qdataimported, -qdatalocal, -qtocdata" on page 121	None.	Marks data as local or imported in 64-bit compilations.
"-qdirectstorage" on page 124	None.	Informs the compiler that a given compilation unit may reference write-through-enabled or cache-inhibited storage.
"-qsimd" on page 269	None.	Controls whether the compiler can automatically take advantage of vector instructions for processors that support them.
"-qfdpr" on page 135	None.	Provides object files with information that the IBM Feedback Directed Program Restructuring (FDPR) performance-tuning utility needs to optimize the resulting executable file.
"-qhot" on page 155	#pragma nosimd, #pragma novector	Performs high-order loop analysis and transformations (HOT) during optimization.
"-qignerrno" on page 160	#pragma options ignerrno	Allows the compiler to perform optimizations that assume `errno` is not modified by system calls.
"-qipa" on page 176	None.	Enables or customizes a class of optimizations known as interprocedural analysis (IPA).
"-qisolated_call" on page 182	#pragma options isolated_call, #pragma isolated_call	Specifies functions in the source file that have no side effects other than those implied by their parameters.
"-qlibansi" on page 208	#pragma options libansi	Assumes that all functions with the name of an ANSI C library function are in fact the system functions.
"-qlibmpi" on page 208	None.	Asserts that all functions with Message Passing Interface (MPI) names are in fact MPI functions and not a user function with different semantics.

Table 17. Optimization and tuning options (continued)

Option name	Equivalent pragma name	Description
"-qmaxmem" on page 220	#pragma options maxmem	Limits the amount of memory that the compiler allocates while performing specific, memory-intensive optimizations to the specified number of kilobytes.
"-qminimaltoc" on page 224	None.	Controls the generation of the table of contents (TOC), which the compiler creates for an executable file in 64-bit compilation mode.
"-O, -qoptimize" on page 227	#pragma options optimize	Specifies whether to optimize code during compilation and, if so, at which level.
"-p, -pg, -qprofile" on page 231	None.	Prepares the object files produced by the compiler for profiling.
"-qpdf1, -qpdf2" on page 236	None.	Tunes optimizations through *profile-directed feedback* (PDF), where results from sample program execution are used to improve optimization near conditional branches and in frequently executed code sections.
"-qprefetch" on page 245	None.	Inserts prefetch instructions automatically where there are opportunities to improve code performance.
"-qprocimported, -qproclocal, -qprocunknown" on page 250	#pragma options procimported, #pragma options proclocal, #pragma options procunkown	Marks functions as local, imported, or unknown in 64-bit compilations.
"-qinline" on page 173	None.	Attempts to inline functions instead of generating calls to those functions, for improved performance.
"-qrestrict (C only)" on page 257	None.	Indicates to the compiler that no other pointers can access the same memory that has been addressed by function parameter pointers.
"-qshowpdf" on page 268	None.	When used with **-qpdf1** and a minimum optimization level of **-O2** at compile and link steps, inserts additional profiling information into the compiled application to collect call and block counts for all procedures in the application.
"-qsmallstack" on page 271	None.	Reduces the size of the stack frame.

Table 17. Optimization and tuning options (continued)

Option name	Equivalent pragma name	Description
"-qsmp" on page 272	None.	Enables parallelization of program code.
"-qstrict" on page 286	#pragma options strict	Ensures that optimizations done by default at optimization levels **-O3** and higher, and, optionally at **-O2**, do not alter the semantics of a program.
"-qstrict_induction" on page 290	None.	Prevents the compiler from performing induction (loop counter) variable optimizations. These optimizations may be unsafe (may alter the semantics of your program) when there are integer overflow operations involving the induction variables.
"-qtune" on page 308	#pragma options tune	Tunes instruction selection, scheduling, and other architecture-dependent performance enhancements to run best on a specific hardware architecture.
"-qunroll" on page 312	#pragma options unroll, #pragma unroll	Controls loop unrolling, for improved performance.
"-qunwind" on page 314	None.	Specifies whether the call stack can be unwound by code looking through the saved registers on the stack.

Linking

Though linking occurs automatically, the options in this category allow you to direct input and output to the linker, controlling how the linker processes your object files.

Table 18. Linking options

Option name	Equivalent pragma name	Description
"-qcrt" on page 117	None.	Specifies whether system startup files are to be linked.
"-e" on page 126	None.	When used together with the **-qmkshrobj**, specifies an entry point for a shared object.
"-L" on page 187	None.	At link time, searches the directory path for library files specified by the **-l** option.

Table 18. Linking options (continued)

Option name	Equivalent pragma name	Description
"-l" on page 186	None.	Searches for the specified library file, lib*key*.so, and then lib*key*.a for dynamic linking, or just for lib*key*.a for static linking.
"-qlib" on page 207	None.	Specifies whether standard system libraries and XL C/C++ libraries are to be linked.
"-R" on page 253	None.	At link time, writes search paths for shared libraries into the executable, so that these directories are searched at program run time for any required shared libraries.
"-qstaticlink" on page 283	None.	Controls how shared and nonshared runtime libraries are linked into an application.

Portability and migration

The options in this category can help you maintain application behavior compatibility on past, current, and future hardware, operating systems and compilers, or help move your applications to an XL compiler with minimal change.

Table 19. Portability and migration options

Option name	Equivalent pragma name	Description
"-qabi_version (C++ only)" on page 87	None.	Specifies the version of the C++ application binary interface (ABI) version used during compilation. This option is provided for compatibility with different levels of GNU C++.
"-qalign" on page 91	#pragma options align, #pragma align	Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data.
"-qgenproto (C only)" on page 152	None.	Produces prototype declarations from K&R function definitions or function definitions with empty parentheses, and displays them to standard output.
"-qpack_semantic" on page 233	None.	Controls the syntax and semantics of the **#pragma pack** directive.
"-qupconv (C only)" on page 315	#pragma options upconv	Specifies whether the `unsigned` specification is preserved when integral promotions are performed.

Compiler customization

The options in this category allow you to specify alternate locations for compiler components, configuration files, standard include directories, and internal compiler operation. You should only need to use these options in specialized installation or testing scenarios.

Table 20. Compiler customization options

Option name	Equivalent pragma name	Description
"-qasm_as" on page 100	None.	Specifies the path and flags used to invoke the assembler in order to handle assembler code in an `asm` assembly statement.
"-B" on page 103	None.	Determines substitute path names for XL C/C++ executables such as the compiler, assembler, linker, and preprocessor.
"-qcomplexgccincl" on page 115	#pragma complexgcc	Specifies whether to use GCC parameter-passing conventions for complex data types (equivalent to enabling **-qfloat=complexgcc**) for selected include files only.
"-qc_stdinc (C only)" on page 118	None.	Changes the standard search location for the XL C header files.
"-qcpp_stdinc (C++ only)" on page 119	None.	Changes the standard search location for the XL C++ header files.
"-F" on page 133	None.	Names an alternative configuration file or stanza for the compiler.
"-qgcc_c_stdinc (C only)" on page 150	None.	Changes the standard search location for the GNU C system header files.
"-qgcc_cpp_stdinc (C++ only)" on page 151	None.	Changes the standard search location for the GNU C++ system header files.
"-qpath" on page 235	None.	Determines substitute path names for XL C/C++ executables such as the compiler, assembler, and linker.
"-qspill" on page 279	#pragma options spill	Specifies the size (in bytes) of the register spill space, the internal program storage areas used by the optimizer for register spills to storage.
"-t" on page 294	None.	Applies the prefix specified by the **-B** option to the designated components.
"-W" on page 321	None.	Passes the listed options to a component that is executed during compilation.

Deprecated options

The compiler still accepts options listed in the following table. Options without an asterisk have been replaced by other options that provide the same functionality. Options with an asterisk are obsolete, or can produce unexpected results and are not guaranteed to perform as previously documented. Use with discretion.

Table 21. Deprecated options

Option name	Replacement option
-Q	-qinline
-qbigdata*	
-qenablevmx	-qsimd
-qhot=simd | nosimd	-qsimd
-qinfo=private	-qreport
-qinfo=reduction	-qreport
-qipa=clonearch | noclonearch	-qtune
-qipa=cloneproc | nocloneproc	-qtune
-qipa=inline | noinline	-qinline
-qipa=pdfname	-qpdf1=pdfname, -qpdf2=pdfname
-qlanglvl=[no]gnu_externtemplate	-qlanglvl=[no]externtemplate

Individual option descriptions

This section contains descriptions of the individual compiler options available in XL C/C++.

For each option, the following information is provided:

Category
> The functional category to which the option belongs is listed here.

Pragma equivalent
> Many compiler options allow you to use an equivalent pragma directive to apply the option's functionality within the source code, limiting the scope of the option's application to a single source file, or even selected sections of code. Where an option supports the **#pragma options** *option_name* and/or **#pragma** *name* form of the directive, this is indicated.

Purpose
> This section provides a brief description of the effect of the option (and equivalent pragmas), and why you might want to use it.

Syntax
> This section provides the syntax for the option, and where an equivalent **#pragma** *name* is supported, the specific syntax for the pragma. Syntax for **#pragma options** *option_name* forms of the pragma is not provided, as this is normally identical to that of the option. Note that you can also use the C99-style _Pragma operator form of any pragma; although this syntax is not provided in the option descriptions. For complete details on pragma syntax, see "Pragma directive syntax" on page 329

Defaults
> In most cases, the default option setting is clearly indicated in the syntax diagram. However, for many options, there are multiple default settings, depending on other compiler options in effect. This section indicates the different defaults that may apply.

Parameters
> This section describes the suboptions that are available for the option and

pragma equivalents, where applicable. For suboptions that are specific to the command-line option or to the pragma directive, this is indicated in the descriptions.

Usage This section describes any rules or usage considerations you should be aware of when using the option. These can include restrictions on the option's applicability, valid placement of pragma directives, precedence rules for multiple option specifications, and so on.

Predefined macros
Many compiler options set macros that are protected (that is, cannot be undefined or redefined by the user). Where applicable, any macros that are predefined by the option, and the values to which they are defined, are listed in this section. A reference list of these macros (as well as others that are defined independently of option setting) is provided in Chapter 6, "Compiler predefined macros," on page 391

Examples
Where appropriate, examples of the command-line syntax and pragma directive use are provided in this section.

-+ (plus sign) (C++ only)
Category

Input control

Pragma equivalent

None.

Purpose

Compiles any file as a C++ language file.

This option is equivalent to the **-qsourcetype=c++** option.

Syntax

▶▶── -+ ──▶◀

Usage

You can use **-+** to compile a file with any suffix other than .a, .o, .so, .S or .s. If you do not use the **-+** option, files must have a suffix of .C (uppercase C), .cc, .cp, .cpp, .cxx, or .c++ to be compiled as a C++ file. If you compile files with suffix .c (lowercase c) without specifying **-+**, the files are compiled as a C language file.

The **-+** option should not be used together with the **-qsourcetype** option.

Predefined macros

None.

Examples

To compile the file `myprogram.cplspls` as a C++ source file, enter:

```
 xlc++ -+ myprogram.cplspls
```

Related information
- "-qsourcetype" on page 277

-# (pound sign)
Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Previews the compilation steps specified on the command line, without actually invoking any compiler components.

When this option is enabled, information is written to standard output, showing the names of the programs within the preprocessor, compiler, and linker that would be invoked, and the default options that would be specified for each program. The preprocessor, compiler, and linker are not invoked.

Syntax

►►── -# ──►◄

Usage

You can use this command to determine the commands and files that will be involved in a particular compilation. It avoids the overhead of compiling the source code and overwriting any existing files, such as .lst files.

This option displays the same information as **-v**, but does not invoke the compiler. The **-#** option overrides the **-v** option.

Predefined macros

None.

Examples

To preview the steps for the compilation of the source file `myprogram.c`, enter:

```
xlc myprogram.c -#
```

Related information
- "-v, -V" on page 316

-q32, -q64

Category

Object code control

Pragma equivalent

None.

Purpose

Selects either 32-bit or 64-bit compiler mode.

Use the **-q32** and **-q64** options, along with the **-qarch** and **-qtune** compiler options, to optimize the output of the compiler to the architecture on which that output will be used.

Syntax

```
►►── -q──┬─32─┬──────────────────────────────────────────────►◄
         └─64─┘
```

Defaults

-q32

Predefined macros

__64BIT__ is defined to 1 when **-q64** is in effect; otherwise, it is undefined.

Examples

To specify that the executable program `testing` compiled from `myprogram.c` is to run on a computer with a 32-bit Power architecture, enter:

```
xlc -o testing myprogram.c -q32 -qarch=ppc
```

Related information
- "Specifying compiler options for architecture-specific, 32-bit or 64-bit compilation" on page 9
- "-qarch" on page 95
- "-qtune" on page 308

-qabi_version (C++ only)

Category

Portability and migration

Pragma equivalent

None.

Purpose

Specifies the version of the C++ application binary interface (ABI) version used during compilation. This option is provided for compatibility with different levels of GNU C++.

Syntax

```
►►── -q─abi_version──=──┬─2─┬──────────────────────────────────────────────►◄
                        └─1─┘
```

Defaults

-qabi_version=2

Parameters

1   Specifies the same C++ ABI behavior as in GNU C++ 3.2.

2   Specifies the same C++ ABI behavior as in GNU C++ 3.4.

Predefined macros

None.

-qaggrcopy
Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Enables destructive copy operations for structures and unions.

Syntax

```
►►── -q─aggrcopy──=──┬─nooverlap─┬──────────────────────────────────────────►◄
                     └─overlap───┘
```

Defaults

-qaggrcopy=nooverlap

Parameters

overlap | nooverlap
nooverlap assumes that the source and destination for structure and union assignments do not overlap, allowing the compiler to generate faster code. **overlap** inhibits these optimizations.

Predefined macros

None.

-qalias

Category

Optimization and tuning

Pragma equivalent

None

Purpose

Indicates whether a program contains certain categories of aliasing or does not conform to C/C++ standard aliasing rules. The compiler limits the scope of some optimizations when there is a possibility that different names are aliases for the same storage location.

Syntax

```
                                    ┌─:──────────┐
                                    │ ┌notypeptr─┐
                                    │ ├restrict──┤
                                    │ ├global────┤
                                    │ ├noallptrs─┤
                                    │ ├ansi──────┤
                                    │ ├noaddrtaken┤
►►── -q─alias──=──┬───▼addrtaken────┬───────────────────────►◄
                    ├noansi────┤
                    ├allptrs───┤
                    ├noglobal──┤
                    ├norestrict┤
                    └typeptr───┘
```
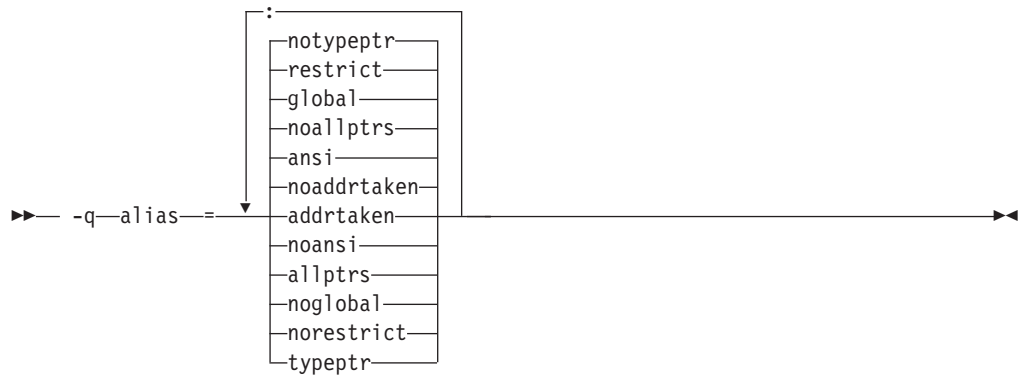
Defaults

- ► **C++** -qalias=noaddrtaken:noallptrs:ansi:global:restrict:notypeptr

- ► **C** **-qalias=noaddrtaken:noallptrs:ansi:global:restrict:notypeptr** for all invocation commands except **cc**.
  -qalias=noaddrtaken:noallptrs:noansi:global:restrict:notypeptr for the **cc** invocation command.

Parameters

addrtaken | noaddrtaken
   When **addrtaken** is in effect, variables are disjoint from pointers unless their address is taken. Any class of variable for which an address has *not* been recorded in the compilation unit will be considered disjoint from indirect access through pointers.

   When **noaddrtaken** is specified, the compiler generates aliasing based on the aliasing rules that are in effect.

allptrs | noallptrs
   When **allptrs** is in effect, pointers are never aliased (this also implies

-qalias=typeptr). Specifying **allptrs** is an assertion to the compiler that no two pointers point to the same storage location. These suboptions are only valid if **ansi** is also specified.

ansi | noansi
When **ansi** is in effect, type-based aliasing is used during optimization, which restricts the lvalues that can be safely used to access a data object. The optimizer assumes that pointers can *only* point to an object of the same type. This suboption has no effect unless you also specify an optimization option.

When **noansi** is in effect, the optimizer makes worst case aliasing assumptions. It assumes that a pointer of a given type can point to an external object or any object whose address is already taken, regardless of type.

global | noglobal
When **global** is in effect, type-based aliasing rules are enabled during IPA link-time optimization across compilation units. Both **-qipa** and **-qalias=ansi** must be enabled for **-qalias=global** to have an effect. Specifying **noglobal** disables type-based aliasing rules.

-qalias=global produces better performance at higher optimization levels and also better link-time performance. If you use **-qalias=global**, it is recommended that you compile as much as possible of the application with the same version of the compiler to maximize the effect of the suboption on performance.

restrict | norestrict
When **restrict** is in effect, optimizations for pointers qualified with the `restrict` keyword are enabled. Specifying **norestrict** disables optimizations for `restrict`-qualified pointers.

-qalias=restrict is independent from other **-qalias** suboptions. Using the **-qalias=restrict** option will usually result in performance improvements for code that uses `restrict`-qualified pointers. Note, however, that using **-qalias=restrict** requires that restricted pointers be used correctly; if they are not, compile-time and runtime failures may result. You can use **norestrict** to preserve compatibility with code compiled with versions of the compiler previous to V9.0.

typeptr | notypeptr
When **typeptr** is in effect, pointers to different types are never aliased. Specifying **typeptr** is an assertion to the compiler that no two pointers of different types point to the same storage location. These suboptions are only valid if **ansi** is also specified.

Usage

-qalias makes assertions to the compiler about the code that is being compiled. If the assertions about the code are false, then the code generated by the compiler may result in unpredictable behaviour when the application is run.

The following are not subject to type-based aliasing:
- Signed and unsigned types. For example, a pointer to a `signed int` can point to an `unsigned int`.
- Character pointer types can point to any type.
- Types qualified as `volatile` or `const`. For example, a pointer to a `const int` can point to an `int`.

Predefined macros

None.

Examples

To specify worst-case aliasing assumptions when compiling `myprogram.c`, enter:

```
xlc myprogram.c -O -qalias=noansi
```

Related information
- "-qipa" on page 176
- "#pragma disjoint" on page 341
- *Type-based aliasing* in the *XL C/C++ Language Reference*
- *The restrict type qualifier* in the *XL C/C++ Language Reference*
- "-qrestrict (C only)" on page 257

-qalign

Category

Portability and migration

Pragma equivalent

#pragma options align, #pragma align

Purpose

Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data.

Syntax

Option syntax

```
►►─ -q─align──=──┬─linuxppc──┬──────────────────────────────────────►◄
                 └─bit_packed─┘
```

Pragma syntax

```
►►──#──pragma──align──(──┬─linuxppc──┬──)──────────────────────────►◄
                         ├─bit_packed─┤
                         └─reset─────┘
```

Defaults

-qalign=power

linuxppc

Parameters

bit_packed
   Bit field data is packed on a bitwise basis without respect to byte boundaries.

linuxppc
> Uses GNU C/C++ alignment rules to maintain binary compatibility with GNU C/C++ objects.

reset (pragma only)
> Discards the current pragma setting and reverts to the setting specified by the previous pragma directive. If no previous pragma was specified, reverts to the command-line or default option setting.

Usage

If you use the **-qalign** option more than once on the command line, the last alignment rule specified applies to the file.

The pragma directives override the **-qalign** compiler option setting for a specified section of program source code. The pragmas affect all aggregate definitions that appear after a given pragma directive; if a pragma is placed inside a nested aggregate, it applies only to the definitions that follow it, not to any containing definitions. Any aggregate variables that are declared use the alignment rule that applied at the point at which the aggregate was *defined*, regardless of pragmas that precede the declaration of the variables. See below for examples.

For a complete explanation of the option and pragma parameters, as well as usage considerations, see "Aligning data" in the *XL C/C++ Optimization and Programming Guide*.

Predefined macros

None.

Examples

The following examples show the interaction of the option and pragmas. Assuming compilation with the command `xlc file2.c`, the following example shows how the pragma affects only an aggregate *definition*, not subsequent declarations of variables of that aggregate type.

```
/* file2.c  The default alignment rule is in effect */

typedef struct A A2;

#pragma options align=bit_packed /*  bit_packed alignment rules are now in effect */
struct A {
int a;
char c;
}; #pragma options align=reset /*  Default alignment rules are in effect again */

struct A A1;  /*  A1 and A3 are aligned using bit_packed alignment rules since   */
A2 A3;        /*  this rule applied when struct A was defined       */
```

Assuming compilation with the command `xlc file.c -qalign=bit_packed`, the following example shows how a pragma embedded in a nested aggregate definition affects only the definitions that follow it.

```
/* file2.c  The default alignment rule in effect is bit_packed  */

struct A {
int a;
#pragma options align=linuxppc /* Applies to B; A is unaffected  */
   struct B {
   char c;
```

```
        double d;
                } BB;    /*  BB uses linuxppc alignment rules  */
} AA;                    /*  AA uses bit_packed alignment rules  /*
```

Related information

- "#pragma pack" on page 360
- "Aligning data" in the *XL C/C++ Optimization and Programming Guide*
- "The __align specifier" in the *XL C/C++ Language Reference*
- "The aligned variable attribute" in the *XL C/C++ Language Reference*
- "The packed variable attribute" in the *XL C/C++ Language Reference*

-qalloca, -ma (C only)

Category

Object code control

Pragma equivalent

#pragma alloca

Purpose

Provides an inline definition of system function `alloca` when it is called from source code that does not include the `alloca.h` header.

The function `void* alloca(size_t size)` dynamically allocates memory, similarly to the standard library function `malloc`. The compiler automatically substitutes calls to the system `alloca` function with an inline built-in function `__alloca` in any of the following cases:

- You include the header file alloca.h
- You compile with **-Dalloca=__alloca**
- You directly call the built-in function using the form `__alloca`

The **-qalloca** and **-ma** options and **#pragma alloca** directive provide the same functionality in C only, if any of the above methods are not used.

Syntax

Option syntax

```
►►─┬─ -q─alloca─┬──────────────────────────────────────────────►◄
   └─ -ma───────┘
```

Pragma syntax

```
►►──#──pragma──alloca──────────────────────────────────────────►◄
```

Defaults

Not applicable.

Usage

If you do not use any of the above-mentioned methods to ensure that calls to `alloca` are replaced with `__alloca`, `alloca` is treated as a user-defined identifier rather than as a built-in function.

Once specified, **#pragma alloca** applies to the rest of the file and cannot be disabled. If a source file contains any functions that you want compiled without **#pragma alloca**, place these functions in a different file.

You may want to consider using a C99 variable length array in place of `alloca`.

Predefined macros

None.

Examples

To compile `myprogram.c` so that calls to the function `alloca` are treated as inline, enter:

```
xlc myprogram.c -qalloca
```

Related information
- "-D" on page 120
- "__alignx" on page 501

-qaltivec
Category

Language element control

Pragma equivalent

None.

Purpose

Enables compiler support for vector data types and operators.

See the *XL C/C++ Language Reference* for complete documentation of vector data types.

Syntax

```
          ┌─noaltivec─┐
►►── -q──┴─altivec──┴───────────────────────────────────►◄
```

Defaults

-qnoaltivec

Usage

This option has effect only when you set or imply **-qarch** to be an architecture that supports vector instructions. Otherwise, the compiler ignores **-qaltivec** and issues a warning message.

Predefined macros

__ALTIVEC__ is defined to 1 and __VEC__ is defined to 10205 when **-qaltivec** is in effect; otherwise, they are undefined.

Examples

To enable compiler support for vector programming, enter the following command:

```
xlc myprogram.c -qarch=ppc64v -qaltivec
```

Related information
- "-qarch"
- "-qsimd" on page 269
- *AltiVec Technology Programming Interface Manual*, available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf

-qarch

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Specifies the processor architecture for which the code (instructions) should be generated.

Syntax

```
►►── -q─arch──=──┬─ppc64grsq─┬──────────────────────────►◄
                 ├─auto──────┤
                 ├─pwr3──────┤
                 ├─pwr4──────┤
                 ├─pwr5──────┤
                 ├─pwr5x─────┤
                 ├─pwr6──────┤
                 ├─pwr6e─────┤
                 ├─pwr7──────┤
                 ├─ppc───────┤
                 ├─ppc64v────┤
                 ├─ppc64─────┤
                 ├─ppcgr─────┤
                 ├─ppc64gr───┤
                 ├─ppc970────┤
                 ├─rs64b─────┤
                 └─rs64c─────┘
```
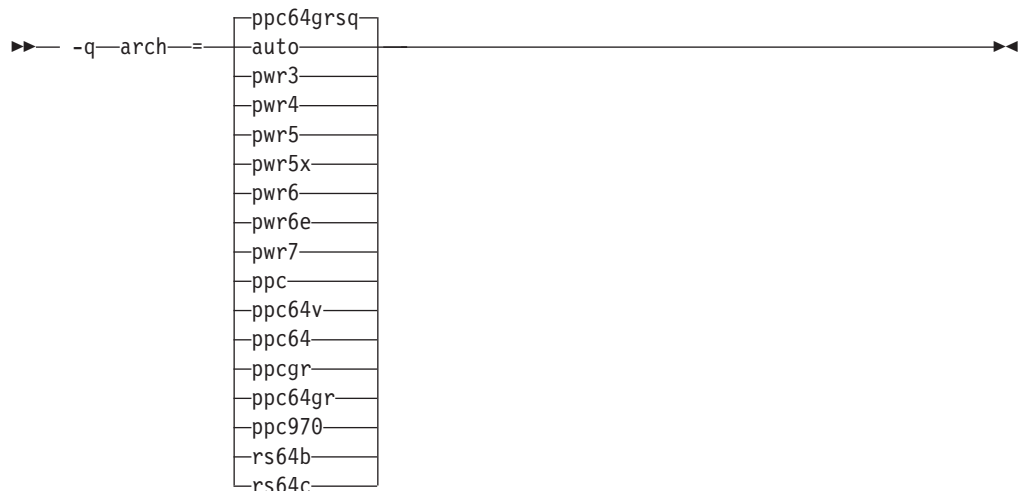
Defaults

- **-qarch=ppc64grsq**
- **-qarch=auto** when **-O4** or **-O5** is in effect.

Parameters

auto
Automatically detects the specific architecture of the compiling machine. It assumes that the execution environment will be the same as the compilation environment. This option is implied if the **-O4** or **-O5** option is set or implied.

pwr3
Produces object code containing instructions that will run on the POWER3™, POWER4™, POWER5™, POWER5+™, POWER6®, POWER7™, or PowerPC® 970 hardware platforms.

pwr4
Produces object code containing instructions that will run on the POWER4, POWER5, POWER5+, POWER6, POWER7, or PowerPC 970 hardware platforms.

pwr5
Produces object code containing instructions that will run on the POWER5, POWER5+, POWER6, or POWER7 hardware platforms.

pwr5x
Produces object code containing instructions that will run on the POWER5+, POWER6, or POWER7 hardware platforms.

pwr6
Produces object code containing instructions that will run on the POWER6 or POWER7 hardware platforms running in POWER6 or POWER7 architected mode.

pwr6e
Produces object code containing instructions that will run on the POWER6 hardware platforms running in POWER6 raw mode.

pwr7
Produces object code containing instructions that will run on the POWER7 hardware platforms.

ppc
In 32-bit mode, produces object code containing instructions that will run on any of the 32-bit PowerPC hardware platforms. This suboption causes the compiler to produce single-precision instructions to be used with single-precision data. Specifying **-qarch=ppc** together with **-q64** silently upgrades the architecture setting to **-qarch=ppc64grsq**.

ppc64
Produces object code that will run on any of the 64-bit PowerPC hardware platforms. This suboption can be selected when compiling in 32-bit mode, but the resulting object code may include instructions that are not recognized or behave differently when run on 32-bit PowerPC platforms.

ppcgr
In 32-bit mode, produces object code for PowerPC processors that support optional graphics instructions. Specifying **-qarch=ppcgr** together with **-q64** silently upgrades the architecture setting to **-qarch=ppc64grsq**.

ppc64gr
> Produces code for any 64-bit PowerPC hardware platform that supports optional graphics instructions.

ppc64grsq
> Produces code for any 64-bit PowerPC hardware platform that supports optional graphics and square root instructions.

ppc64v
> Generates instructions for generic PowerPC chips with vector processors, such as the PowerPC 970. Valid in 32-bit or 64-bit mode.

ppc970
> Generates instructions specific to the PowerPC 970 architecture.

rs64b
> Produces object code that will run on RS64II platforms.

rs64c
> Produces object code that will run on RS64III platforms.

Usage

All PowerPC machines share a common set of instructions, but may also include additional instructions unique to a given processor or processor family. Using the **-qarch** option to target a specific architecture for the compilation results in code that may not run on other architectures, but provides the best performance for the selected architecture. If you want maximum performance on a specific architecture and will not be using the program on other architectures, use the appropriate architecture option. If you want to generate code that can run on more than one architecture, specify a **-qarch** suboption that supports a group of architectures. Table 22 shows the features supported by the different processor architectures and their representative **-qarch** suboptions:

Table 22. Feature support in processor architectures

Architecture	Graphics support	Square root support	64-bit support	Vector processing support
rs64b	yes	yes	yes	no
rs64c	yes	yes	yes	no
pwr3	yes	yes	yes	no
pwr4	yes	yes	yes	no
pwr5	yes	yes	yes	no
pwr5x	yes	yes	yes	no
ppc	no	no	no	no
ppc64	no	no	yes	no
ppc64gr	yes	no	yes	no
ppc64grsq	yes	yes	yes	no
ppc64v	yes	yes	yes	VMX
ppc970	yes	yes	yes	VMX
pwr6	yes	yes	yes	VMX
pwr6e	yes	yes	yes	VMX
pwr7	yes	yes	yes	VMX, VSX

Note: Vector Multimedia Extension (VMX) and Vector Scalar Extension (VSX) are processor instructions for vector processing.

For any given **-qarch** setting, the compiler defaults to a specific, matching **-qtune** setting, which can provide additional performance improvements. Alternatively, if you specify **-qarch** with a group argument, you can specify **-qtune** as either **auto** or provide a specific architecture in the group. For detailed information on using **-qarch** and **-qtune** together, see "-qtune" on page 308.

Specifying **-q64** changes the effective **-qarch** setting as follows:

Original -qarch setting	Effective setting when -q64 is specified
ppc	ppc64grsq
ppcgr	ppc64grsq

For a given application program, make sure that you specify the same **-qarch** setting when you compile each of its source files.

Predefined macros

See "Macros related to architecture settings" on page 396 for a list of macros that are predefined by **-qarch** suboptions.

Examples

To specify that the executable program `testing` compiled from `myprogram.c` is to run on a computer with a 32-bit PowerPC architecture, enter:

```
xlc -o testing myprogram.c -q32 -qarch=ppc
```

Related information
- "Specifying compiler options for architecture-specific, 32-bit or 64-bit compilation" on page 9
- -qfloat
- -qprefetch
- "-qtune" on page 308
- "-q32, -q64" on page 87
- "Macros related to architecture settings" on page 396
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*

-qasm

Category

Language element control

Pragma equivalent

None.

Purpose

Controls the interpretation of and subsequent generation of code for assembler language extensions.

When **-qasm** is in effect, the compiler generates code for assembly statements in the source code. Suboptions specify the syntax used to interpret the content of the assembly statement.

Note: The system assembler program must be available for this command to have effect.

Syntax

-qasm syntax — C

-qasm syntax — C++

Defaults

-qasm=gcc

Parameters

gcc
Instructs the compiler to recognize the extended GCC syntax and semantics for assembly statements.

▶ `C++` **stdcpp**
Reserved for possible future use.

Specifying **-qasm** without a suboption is equivalent to specifying the default.

Usage

▶ `C` The token `asm` is not a C language keyword. Therefore, at language levels **stdc89** and **stdc99**, which enforce strict compliance to the C89 and C99 standards, respectively, the option **-qkeyword=asm** must also be specified to compile source that generates assembly code. At all other language levels, the token `asm` is treated as a keyword unless the option **-qnokeyword=asm** is in effect. In C, the compiler-specific variants `__asm` and `__asm__` are keywords at all language levels and cannot be disabled.

▶ `C++` The tokens `asm`, `__asm`, and `__asm__` are keywords at all language levels. Suboptions of **-qnokeyword=*token*** can be used to disable each of these reserved words individually.

For detailed information on the syntax and semantics of inline `asm` statements, see "Inline assembly statements" in the *XL C/C++ Language Reference*.

Predefined macros

- ▶ `C` __IBM_GCC_ASM is predefined to 1 when `asm` is recognized as a keyword and assembler code is generated; that is, at all language levels except **stdc89 | stdc99**, or when **-qkeyword=asm** is in effect, and when **-qasm[=gcc]** is

in effect. It is predefined to 0 when `asm` is recognized as a keyword but assembler code is not generated; that is, at all language levels except **stdc89 | stdc99**, or when **-qkeyword=asm** is in effect, and when **-qnoasm** is in effect. It is undefined when the **stdc89 | stdc99** language level or **-qnokeyword=asm** is in effect.

- ▶ C++ ◀ __IBM_GCC_ASM is predefined to 1 when `asm` is recognized as a keyword and assembler code is generated; that is, at all language levels, and when **-qasm[=gcc]** is in effect. It is predefined to 0 when `asm` is recognized as a keyword but assembler code is not generated; that is, at all language levels, and when **-qnoasm[=gcc]** is in effect. It is undefined when **-qnoasm=stdcpp** is in effect. __IBM_STDCPP_ASM is predefined to 0 when **-qnoasm=stdcpp** is in effect; otherwise it is undefined.

Examples

The following code snippet shows an example of the GCC conventions for `asm` syntax in inline statements:

```
int a, b, c;
int main() {
    asm("add %0, %1, %2" : "=r"(a) : "r"(b), "r"(c) );
}
```

Related information
- "-qasm_as"
- "-qlanglvl" on page 188
- "-qkeyword" on page 185
- "Inline assembly statements" in the *XL C/C++ Language Reference*
- "Keywords for language extensions"

-qasm_as
Category

Compiler customization

Pragma equivalent

None.

Purpose

Specifies the path and flags used to invoke the assembler in order to handle assembler code in an `asm` assembly statement.

Normally the compiler reads the location of the assembler from the configuration file; you can use this option to specify an alternate assembler program and flags to pass to that assembler.

Syntax

```
►►── -q──asm_as──=──┬─path────────────────┬──────────────►◄
                    └─"──path──┬───────┬──"─┘
                              └─flags─┘
```

Defaults

By default, the compiler invokes the assembler program defined for the **as** command in the compiler configuration file.

Parameters

path
> The full path name of the assembler to be used.

flags
> A space-separated list of options to be passed to the assembler for assembly statements. Quotation marks must be used if spaces are present.

Predefined macros

None.

Examples

To instruct the compiler to use the assembler program at /bin/as when it encounters inline assembler code in myprogram.c, enter:

```
xlc myprogram.c -qasm_as=/bin/as
```

To instruct the compiler to pass some additional options to the assembler at /bin/as for processing inline assembler code in myprogram.c, enter:

```
xlc myprogram.c -qasm_as="/bin/as -a64 -l a.lst"
```

Related information

- "-qasm" on page 98

-qassert

Category

Optimization and tuning

Pragma equivalent

None

Purpose

Provides information about the characteristics of the files that can help to fine-tune optimizations.

Syntax

Option:

Defaults

-qassert=norefalign

Parameters

refalign | **norefalign**
> Specifies that all pointers inside the compilation unit only point to data that is naturally aligned according to the length of the pointer types. With this assertion, the compiler might generate more efficient code. This assertion is particularly useful when you target a SIMD architecture with **-qhot=level=0** or **-qhot=level=1** with **-qsimd=auto**.

-qattr

Category

Listings, messages, and compiler information

Pragma equivalent

#pragma options [no]attr

Purpose

Produces a compiler listing that includes the attribute component of the attribute and cross-reference section of the listing.

Syntax

```
              ┌─noattr─────────┐
►►─── -q──────┼─attr──────────┼─────────────────────────────────────────►◄
                   └─=─full─┘
```

Defaults

-qnoattr

Parameters

full
> Reports all identifiers in the program. If you specify **attr** without this suboption, only those identifiers that are used are reported.

Usage

If **-qattr** is specified after **-qattr=full**, it has no effect; the full listing is produced.

This option does not produce a cross-reference listing unless you also specify **-qxref**.

The **-qnoprint** option overrides this option.

Predefined macros

None.

Examples

To compile the program `myprogram.c` and produce a compiler listing of all identifiers, enter:

```
xlc myprogram.c -qxref -qattr=full
```

Related information
- "-qprint" on page 248
- "-qxref" on page 325

-B

Category

Compiler customization

Pragma equivalent

None.

Purpose

Determines substitute path names for XL C/C++ executables such as the compiler, assembler, linker, and preprocessor.

You can use this option if you want to keep multiple levels of some or all of the XL C/C++ executables and have the option of specifying which one you want to use. However, it is recommended that you use the **-qpath** option to accomplish this instead.

Syntax

```
►►─── -B ──────────────────────────────────────────────►◄
          └─prefix─┘
```

Defaults

The default paths for the compiler executables are defined in the compiler configuration file.

Parameters

prefix
> Defines part of a path name for programs you can name with the **-t** option. You must add a slash (/). If you specify the **-B** option without the *prefix*, the default prefix is /lib/o.

Usage

The **-t** option specifies the programs to which the **-B** prefix name is to be appended; see "-t" on page 294 for a list of these. If you use the **-B** option without **-t***programs*, the prefix you specify applies to all of the compiler executables.

The **-B** and **-t** options override the **-F** option.

Predefined macros

None.

Examples

In this example, an earlier level of the compiler components is installed in the default installation directory. To test the upgraded product before making it available to everyone, the system administrator restores the latest installation image under the directory /home/jim and then tries it out with commands similar to:

```
xlc -tcbI -B/home/jim/opt/ibmcmp/vacpp/11.1/bin/ test_suite.c
```

Once the upgrade meets the acceptance criteria, the system administrator installs it in the default installation directory.

Related information
- "-qpath" on page 235
- "-t" on page 294
- "Invoking the compiler" on page 1

-qbitfields
Category

Floating-point and integer control

Pragma equivalent

None.

Purpose

Specifies whether bit fields are signed or unsigned.

Syntax

```
►►─── -q─bitfields─=─┬─signed───┬─────────────────────────────────►◄
                     └─unsigned─┘
```

Defaults

-qbitfields=signed

Parameters

signed
    Bit fields are signed.

unsigned
    Bit fields are unsigned.

Predefined macros

None.

-c

Category

Output control

Pragma equivalent

None.

Purpose

Prevents the completed object from being sent to the linker. With this option, the output is a .o file for each source file.

Syntax

▶▶── -c ──▶◀

Defaults

By default, the compiler invokes the linker to link object files into a final executable.

Usage

When this option is in effect, the compiler creates an output object file, *file_name*.o, for each valid source file, such as *file_name*.c, *file_name*.i, *file_name*.C, *file_name*.cpp. You can use the **-o** option to provide an explicit name for the object file.

The **-c** option is overridden if the **-E**, **-P**, or **-qsyntaxonly** options are specified.

Predefined macros

None.

Examples

To compile myprogram.c to produce an object file myprogram.o, but no executable file, enter the command:

```
xlc myprogram.c -c
```

To compile myprogram.c to produce the object file new.o and no executable file, enter:

```
xlc myprogram.c -c -o new.o
```

Related information
- "-E" on page 127
- "-o" on page 226
- "-P" on page 232
- "-qsyntaxonly (C only)" on page 293

-C, -C!

Category

Output control

Pragma equivalent

None.

Purpose

When used in conjunction with the **-E** or **-P** options, preserves or removes comments in preprocessed output.

When **-C** is in effect, comments are preserved. When **-C!** is in effect, comments are removed.

Syntax

```
►►──┬─ -C ──┬──────────────────────────────────────────────►◄
    └─ -C! ─┘
```

Defaults

-C

Usage

The **-C** option has no effect without either the **-E** or the **-P** option. If **-E** is specified, continuation sequences are preserved in the output. If **-P** is specified, continuation sequences are stripped from the output, forming concatenated output lines.

You can use the **-C!** option to override the **-C** option specified in a default makefile or configuration file.

Predefined macros

None.

Examples

To compile `myprogram.c` to produce a file `myprogram.i` that contains the preprocessed program text including comments, enter:

```
xlc myprogram.c -P -C
```

Related information
- "-E" on page 127
- "-P" on page 232

-qcache

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

When specified with **-O4**, **-O5**, or **-qipa**, specifies the cache configuration for a specific execution machine.

If you know the type of execution system for a program, and that system has its instruction or data cache configured differently from the default case, use this option to specify the exact cache characteristics. The compiler uses this information to calculate the benefits of cache-related optimizations.

Syntax

```
               ┌─ : ─────────────────────────┐
               ▼                              │
►►─ -q─cache─=─┬─assoc─=─number─┐─────────────────────────────────►◄
               ├─auto──────────┤
               ├─cost─=─cycles─┤
               ├─level─=─┬─1─┐─┤
               │         ├─2─┤ │
               │         └─3─┘ │
               ├─line─=─bytes──┤
               ├─size─=─Kbytes─┤
               └─type─=─┬─c─┐──┘
                        ├─d─┤
                        └─i─┘
```
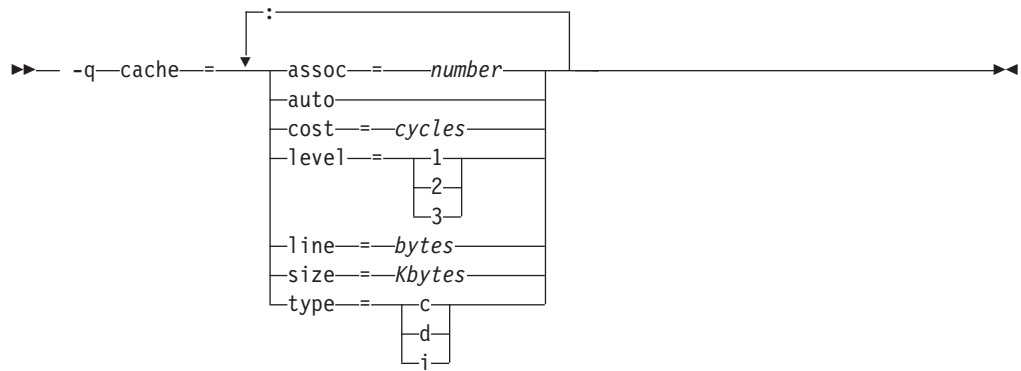
Defaults

Automatically determined by the setting of the **-qtune** option.

Parameters

assoc
Specifies the set associativity of the cache.

number
Is one of:

0   Direct-mapped cache

1   Fully associative cache

N>1 n-way set associative cache

auto
Automatically detects the specific cache configuration of the compiling machine. This assumes that the execution environment will be the same as the compilation environment.

cost
Specifies the performance penalty resulting from a cache miss.

cycles

level
    Specifies the level of cache affected. If a machine has more than one level of cache, use a separate **-qcache** option.

level
    Is one of:

    1   Basic cache

    2   Level-2 cache or, if there is no level-2 cache, the table lookaside buffer (TLB)

    3   TLB

line
    Specifies the line size of the cache.

bytes
    An integer representing the number of bytes of the cache line.

size
    Specifies the total size of the cache.

Kbytes
    An integer representing the number of kilobytes of the total cache.

type
    Specifies that the settings apply to the specified *cache_type*.

cache_type
    Is one of:

    c   Combined data and instruction cache

    d   Data cache

    i   Instruction cache

Usage

The **-qtune** setting determines the optimal default **-qcache** settings for most typical compilations. You can use the **-qcache** to override these default settings. However, if you specify the wrong values for the cache configuration, or run the program on a machine with a different configuration, the program will work correctly but may be slightly slower.

You must specify **-O4**, **-O5**, or **-qipa** with the **-qcache** option.

Use the following guidelines when specifying **-qcache** suboptions:
- Specify information for as many configuration parameters as possible.
- If the target execution system has more than one level of cache, use a separate **-qcache** option to describe each cache level.
- If you are unsure of the exact size of the cache(s) on the target execution machine, specify an estimated cache size on the small side. It is better to leave some cache memory unused than it is to experience cache misses or page faults from specifying a cache size larger than actually present.
- The data cache has a greater effect on program performance than the instruction cache. If you have limited time available to experiment with different cache configurations, determine the optimal configuration specifications for the data cache first.

- If you specify the wrong values for the cache configuration, or run the program on a machine with a different configuration, program performance may degrade but program output will still be as expected.
- The **-O4** and **-O5** optimization options automatically select the cache characteristics of the compiling machine. If you specify the **-qcache** option together with the **-O4** or **-O5** options, the option specified last takes precedence.

Predefined macros

None.

Examples

To tune performance for a system with a combined instruction and data level-1 cache, where cache is 2-way associative, 8 KB in size and has 64-byte cache lines, enter:

```
xlc -O4 -qcache=type=c:level=1:size=8:line=64:assoc=2 file.c
```

Related information
- "-qcache" on page 106
- "-O, -qoptimize" on page 227
- "-qtune" on page 308
- "-qipa" on page 176
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*

-qchars

Category

Floating-point and integer control

Pragma equivalent

#pragma options chars, #pragma chars

Purpose

Determines whether all variables of type char are treated as either signed or unsigned.

Syntax

Option syntax

```
►►── -q──chars──=──┬─unsigned─┬──────────────────────────────────►◄
                   └─signed───┘
```

Pragma syntax

```
►►──#──pragma──chars──(──┬─unsigned─┬──)───────────────────────►◄
                         └─signed───┘
```

Defaults

-qchars=unsigned

Parameters

unsigned
Variables of type `char` are treated as `unsigned char`.

signed
Variables of type `char` are treated as `signed char`.

Usage

Regardless of the setting of this option or pragma, the type of `char` is still considered to be distinct from the types `unsigned char` and `signed char` for purposes of type-compatibility checking or C++ overloading.

The pragma must appear before any source statements. If the pragma is specified more than once in the source file, the first one will take precedence. Once specified, the pragma applies to the entire file and cannot be disabled; if a source file contains any functions that you want to compile without **#pragma chars**, place these functions in a different file.

Predefined macros

- _CHAR_SIGNED and __CHAR_SIGNED__ are defined to 1 when **signed** is in effect; otherwise, it is undefined.
- _CHAR_UNSIGNED and _CHAR_UNSIGNED__ are defined to 1 when **unsigned** is in effect; otherwise, they are undefined.

Examples

To treat all `char` types as `signed` when compiling `myprogram.c`, enter:

```
xlc myprogram.c -qchars=signed
```

-qcheck

Category

Error checking and debugging

Pragma equivalent

#pragma options [no]check

Purpose

Generates code that performs certain types of runtime checking.

If a violation is encountered, a runtime error is raised by sending a SIGTRAP signal to the process. Note that the runtime checks may result in slower application execution.

Syntax

```
                     ┌─nocheck─────────────────────────────────────┐
►►── -q───┴─check───┬─────────────────────────────────────────┬──►◄
                    │              ┌─:─────────┐                │
                    │              │  ┌─all──────┐            │
                    └─=──────▼─────┼─bounds────┤──────────┘
                                   ├─nobounds──┤
                                   ├─divzero───┤
                                   ├─nodivzero─┤
                                   ├─nullptr───┤
                                   └─nonullptr─┘
```
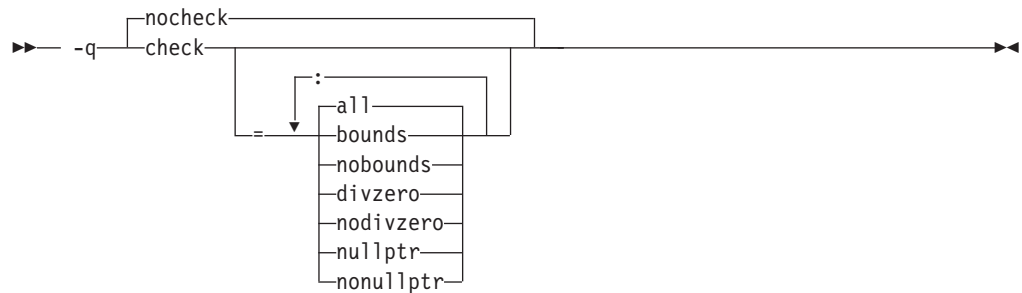
Defaults

-qnocheck

Parameters

all
   Enables all suboptions.

bounds | nobounds
   Performs runtime checking of addresses for subscripting within an object of
   known size. The index is checked to ensure that it will result in an address that
   lies within the bounds of the object's storage. A trap will occur if the address
   does not lie within the bounds of the object.

   This suboption has no effect on accesses to a variable length array.

divzero | nodivzero
   Performs runtime checking of integer division. A trap will occur if an attempt
   is made to divide by zero.

nullptr | nonullptr
   Performs runtime checking of addresses contained in pointer variables used to
   reference storage. The address is checked at the point of use; a trap will occur
   if the value is less than 512.

Specifying the **-qcheck** option with no suboptions is equivalent to **-qcheck=all**.

Usage

You can specify the **-qcheck** option more than once. The suboption settings are
accumulated, but the later suboptions override the earlier ones.

You can use the **all** suboption along with the **no...** form of one or more of the other
options as a filter. For example, using:

```
xlc myprogram.c -qcheck=all:nonullptr
```

provides checking for everything except for addresses contained in pointer
variables used to reference storage. If you use **all** with the **no...** form of the
suboptions, **all** should be the first suboption.

Predefined macros

None.

Examples

The following code example shows the effect of **-qcheck=nullptr:bounds**:

```
void func1(int* p) {
  *p = 42;            /* Traps if p is a null pointer */
}

void func2(int i) {
  int array[10];
  array[i] = 42;     /* Traps if i is outside range 0 - 9 */
}
```

The following code example shows the effect of **-qcheck=divzero**:

```
void func3(int a, int b) {
  a / b;             /* Traps if b=0  */
}
```

-qcinc (C++ only)
Category

Input control

Pragma equivalent

None.

Purpose

Places an `extern "C" { }` wrapper around the contents of include files located in a specified directory.

Syntax

```
          ┌─nocinc───────────────────┐
►►── -q───┴─cinc──=─directory_path────┴──────────────────────────────►◄
```

Defaults

-qnocinc

Parameters

directory_path
 The directory where the include files to be wrapped with an `extern "C"` linkage specifier are located.

Predefined macros

None.

Examples

Assume your application `myprogram.C` includes header file `foo.h`, which is located in directory `/usr/tmp` and contains the following code:

```
int foo();
```

Compiling your application with:

```
xlc++ myprogram.C -qcinc=/usr/tmp
```

will include header file foo.h into your application as:

```
extern "C" {
int foo();
}
```

-qcommon

Category

Object code control

Pragma equivalent

None.

Purpose

Controls where uninitialized global variables are allocated.

When **-qcommon** is in effect, uninitialized global variables are allocated in the common section of the object file. When **-qnocommon** is in effect, uninitialized global variables are initialized to zero and allocated in the data section of the object file.

Syntax


```
►►── -q──┬──common───┬──────────────────────────────────────►◄
         └─nocommon──┘
```

Defaults

- ▶ C ◀ **-qcommon** except when **-qmkshrobj** is specified; **-qnocommon** when **-qmkshrobj** is specified.
- ▶ C++ ◀ **-qnocommon**

Usage

This option does not affect static or automatic variables, or the declaration of structure or union members.

This option is overridden by the common|nocommon and section variable attributes. See "The common and nocommon variable attribute" and "The section variable attribute" in the *XL C/C++ Language Reference*.

Predefined macros

None.

Examples

In the following declaration, where a and b are global variables:

```
int a, b:
```

Compiling with **-qcommon** produces the equivalent of the following assembly code:

```
.comm _a,4
.comm _b,4
```

Compiling with **-qnocommon** produces the equivalent of the following assembly code:

```
        .globl _a
.data
.zerofill __DATA, __common, _a, 4, 2
        .globl _b
.data
.zerofill __DATA, __common, _b, 4, 2
```

Related information
- "-qmkshrobj" on page 224
- "The common and nocommon variable attribute" in the *XL C/C++ Language Reference*
- "The section variable attribute" in the *XL C/C++ Language Reference*

-qcompact
Category

Optimization and tuning

Pragma equivalent

#pragma options [no]compact

Purpose

Avoids optimizations that increase code size.

Code size is reduced by inhibiting optimizations that replicate or expand code inline, such as inlining or loop unrolling. Execution time may increase.

Syntax

```
                ┌─nocompact─┐
►►── -q────────┴─compact───┴─────────────────────────────────────────────►◄
```

Defaults

-qnocompact

Usage

This option only has an effect when specified with an optimization option.

Predefined macros

__OPTIMIZE_SIZE__ is predefined to 1 when **-qcompact** and an optimization level are in effect. Otherwise, it is undefined.

Examples

To compile `myprogram.c`, instructing the compiler to reduce code size whenever possible, enter:

```
xlc myprogram.c -O -qcompact
```

-qcomplexgccincl

Category

Compiler customization

Pragma equivalent

None.

Purpose

Specifies whether to use GCC parameter-passing conventions for complex data types (equivalent to enabling **-qfloat=complexgcc**) for selected include files only.

When **-qcomplexgccincl** is in effect, the compiler internally wraps **#pragma complexgcc(on)** and **#pragma complexgcc(pop)** directives around the files located in specified directories. When **-qnocomplexgccincl** is in effect, include files found in the specified directories are not wrapped by these directives.

You can also use the pragma directives to enable or disable GCC parameter-passing conventions for complex data types for selected files or sections of code.

Syntax

Option syntax

```
►►─── -q───┬─complexgccincl───┬───=──directory_path────────────────────►◄
           └─nocomplexgccincl─┘
```

Pragma syntax

```
►►──#──pragma──complexgcc──(──┬──on───┬──)───────────────────────────►◄
                             ├──off──┤
                             └──pop──┘
```

Defaults

By default, files located in the standard directories for the XL C/C++ and GCC header files are wrapped with **#pragma complexgcc** directives. For a list of these, see "Directory search sequence for include files" on page 12.

Parameters

directory_path **(option only)**
> The directory in which the include files to be wrapped with **#pragma complexgcc** directives are located. If you do not specify a *directory_path*, the compiler assumes the default directories listed above.

on (pragma only)

   Sets **-qfloat=gccomplex** for the code that follows it. This instructs the compiler to use the GCC conventions for passing and returning parameters of complex type, by using general purpose registers.

off (pragma only)

   Sets **-qfloat=nogccomplex** for the code that follows it. This instructs the compiler to use AIX® conventions for passing and returning parameters of complex type, by using floating-point registers.

pop (pragma only)

   Discards the current pragma setting and reverts to the setting specified by the previous pragma directive. If no previous pragma was specified, reverts to the command-line or default option setting.

Usage

The current setting of the pragma affects only functions declared or defined while the setting is in effect. It does not affect other functions.

Calling functions through pointers to functions will always use the convention set by the **-qfloat=[no]complexgcc** command-line option in effect. An error will result if you mix and match functions that pass complex values by value or return complex values. For example, assume the following code is compiled with **-qfloat=nocomplexgcc**:

```
#pragma complexgcc(on)
void p (_Complex double x) {}

#pragma complexgcc(pop)
typedef void (*fcnptr) (_Complex double);

int main() {
    fcnptr ptr = p; /* error: function pointer is -qfloat=nocomplexgcc;
                             function is -qfloat=complexgcc */
}
```

Predefined macros

None.

Related information
* "-qfloat" on page 137

-qcpluscmt (C only)
Category

Language element control

Pragma equivalent

None.

Purpose

Enables recognition of C++-style comments in C source files.

Syntax

```
         ┌─cpluscmt───┐
►►── -q───┴─nocpluscmt─┘──────────────────────────────────────────────────►◄
```
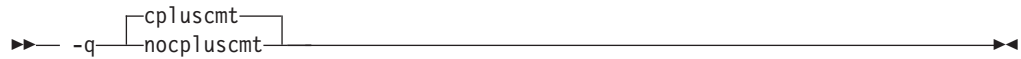
Defaults

- **-qcpluscmt** when the **xlc** or **c99** and related invocations are used, or when the **stdc99** | **extc99** language level is in effect.
- **-qnocpluscmt** for all other invocation commands and language levels.

Predefined macros

__C99_CPLUSCMT is predefined to 1 when **-qcpluscmt** is in effect; otherwise, it is undefined.

Examples

To compile myprogram.c so that C++ comments are recognized as comments, enter:

```
xlc myprogram.c -qcpluscmt
```

Note that // comments are *not* part of C89. The result of the following valid C89 program will be incorrect:

```
main() {
  int i = 2;
  printf("%i\n", i //* 2 */
                 + 1);
}
```

The correct answer is 2 (2 divided by 1). When **-qcpluscmt** is in effect (as it is by default), the result is 3 (2 plus 1).

Related information
- "-C, -C!" on page 106
- "-qlanglvl" on page 188
- "Comments" in the *XL C/C++ Language Reference*

-qcrt

Category

Linking

Pragma equivalent

None.

Purpose

Specifies whether system startup files are to be linked.

When **-qcrt** is in effect, the system startup routines are automatically linked. When **-qnocrt** is in effect, the system startup files are not used at link time; only the files specified on the command line with the **-l** flag will be linked.

This option can be used in system programming to disable the automatic linking of the startup routines provided by the operating system.

Syntax

```
►►── -q──┬─crt──┬──────────────────────────────────────────────────►◄
         └─nocrt─┘
```

Defaults

-qcrt

Predefined macros

None.

Related information
- "-qlib" on page 207

-qc_stdinc (C only)
Category

Compiler customization

Pragma equivalent

None.

Purpose

Changes the standard search location for the XL C header files.

Syntax

```
                            ┌──:──────┐
                            │         │
►►── -q──c_stdinc──=──┬───┬──▼─directory_path─┬──────┬──────────────►◄
                      └─"─┘                   └──"──┘
```

Defaults

By default, the compiler searches the directory specified in the configuration file for the XL C header files (this is normally /opt/ibmcmp/vacpp/11.1/include/).

Parameters

directory_path
  The path for the directory where the compiler should search for the XL C header files. The *directory_path* can be a relative or absolute path. You can surround the path with quotation marks to ensure it is not split up by the command line.

Usage

This option allows you to change the search paths for specific compilations. To permanently change the default search paths for the XL C headers, you use a configuration file to do so; see "Directory search sequence for include files" on page 12 for more information.

If this option is specified more than once, only the last instance of the option is used by the compiler.

This option is ignored if the **-qnostdinc** option is in effect.

Predefined macros

None.

Examples

To override the default search path for the XL C headers with `mypath/headers1` and `mypath/headers2`, enter:

```
xlc myprogram.c -qc_stdinc=mypath/headers1:mypath/headers2
```

Related information
- "-qgcc_c_stdinc (C only)" on page 150
- "-qstdinc" on page 285
- "-qinclude" on page 162
- "Directory search sequence for include files" on page 12
- "Specifying compiler options in a configuration file" on page 7

-qcpp_stdinc (C++ only)
Category

Compiler customization

Pragma equivalent

None.

Purpose

Changes the standard search location for the XL C++ header files.

Syntax

Defaults

By default, the compiler searches the directory specified in the configuration file for the XL C++ header files (this is normally /opt/ibmcmp/vacpp/11.1/include/).

Parameters

directory_path

The path for the directory where the compiler should search for the XL C++ header files. The *directory_path* can be a relative or absolute path. You can surround the path with quotation marks to ensure it is not split up by the command line.

Usage

This option allows you to change the search paths for specific compilations. To permanently change the default search paths for the XL C++ headers, you use a configuration file to do so; see "Directory search sequence for include files" on page 12 for more information.

If this option is specified more than once, only the last instance of the option is used by the compiler.

This option is ignored if the **-qnostdinc** option is in effect.

Predefined macros

None.

Examples

To override the default search path for the XL C++ headers with `mypath/headers1` and `mypath/headers2`, enter:

```
xlc++ myprogram.C -qcpp_stdinc=mypath/headers1:mypath/headers2
```

Related information

- "-qgcc_cpp_stdinc (C++ only)" on page 151
- "-qstdinc" on page 285
- "-qinclude" on page 162
- "Directory search sequence for include files" on page 12
- "Specifying compiler options in a configuration file" on page 7

-D

Category

Language element control

Pragma equivalent

None.

Purpose

Defines a macro as in a `#define` preprocessor directive.

Syntax

```
►►─ -D─name─┬─────────────────┬─────────────────────────────►◄
            └─=─definition─┘
```

Defaults

Not applicable.

Parameters

name
> The macro you want to define. -D*name* is equivalent to `#define` *name*. For example, -DCOUNT is equivalent to `#define COUNT`.

definition
> The value to be assigned to *name*. -D*name=definition* is equivalent to `#define name definition`. For example, -DCOUNT=100 is equivalent to `#define COUNT 100`.

Usage

Using the `#define` directive to define a macro name already defined by the **-D** option will result in an error condition.

The **-U***name* option, which is used to undefine macros defined by the **-D** option, has a higher precedence than the **-D***name* option.

Predefined macros

The compiler configuration file uses the **-D** option to predefine several macro names for specific invocation commands. For details, see the configuration file for your system.

Examples

To specify that all instances of the name `COUNT` be replaced by 100 in `myprogram.c`, enter:

```
xlc myprogram.c -DCOUNT=100
```

Related information
- "-U" on page 311
- Chapter 6, "Compiler predefined macros," on page 391

-qdataimported, -qdatalocal, -qtocdata
Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Marks data as local or imported in 64-bit compilations.

Local variables are statically bound with the functions that use them. You can use the **-qdatalocal** option to name variables that the compiler can assume are local. Alternatively, you can use the **-qtocdata** option to instruct the compiler to assume all variables are local.

Imported variables are dynamically bound with a shared portion of a library. You can use the **-qdataimported** option to name variables that the compiler can assume are imported. Alternatively, you can use the **-qnotocdata** option to instruct the compiler to assume all variables are imported.

Syntax

```
                   ┌─notocdata───────┐
                   │ ┌─dataimported─┐ │
►►── -q ──┬────────┴─┴──────────────┴─┴──────────────────────────────►◄
          │                    ┌──────:──────┐
          │              └─ = ─▼─variable_name─┘
          │
          └─tocdata──────────────────┐
          └─datalocal──┬─────────────┘
                       │      ┌──:──┐
                       └─ = ──▼─variable_name─┘
```
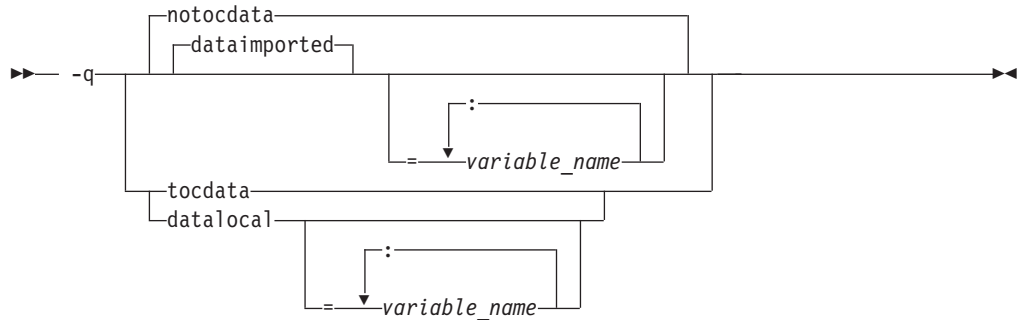
Defaults

-qdataimported or **-qnotocdata**: The compiler assumes all variables are imported.

Parameters

variable_name
>   The name of a variable that the compiler should assume is local or imported (depending on the option specified).
>
>   ▶ `C++` Names must be specified using their mangled names. To obtain C++ mangled names, compile your source to object files only, using the **-c** compiler option, and use the **nm** operating system command on the resulting object file. (See also "Name mangling" in the *XL C/C++ Language Reference* for details on using the `extern "C"` linkage specifier on declarations to prevent name mangling.)
>
>   Specifying **-qdataimported** without any *variable_name* is equivalent to **-qnotocdata**: all variables are assumed to be imported. Specifying **-qdatalocal** without any *variable_name* is equivalent to **-qtocdata**: all variables are assumed to be local.

Usage

These options apply to 64-bit compilations only.

If any variables that are marked as local are actually imported, incorrect code may be generated and performance may decrease.

If you specify any of these options with no variables, the last option specified is used. If you specify the same variable name on more than one option specification, the last one is used.

Predefined macros

None.

-qdbxextra (C only)

Category

Error checking and debugging

Pragma equivalent

#pragma options dbxextra

Purpose

When used with the **-g** option, specifies that debugging information is generated for unreferenced `typedef` declarations, `struct`, `union`, and `enum` type definitions.

To minimize the size of object and executable files, the compiler only includes information for `typedef` declarations, `struct`, `union`, and `enum` type definitions that are referenced by the program. When you specify the **-qdbxextra** option, debugging information is included in the symbol table of the object file. This option is equivalent to the **-qsymtab=unref** option.

Syntax

```
         ┌─nodbxextra─┐
►►── -q───┴─dbxextra──┴──────────────────────────────────────────►◄
```

Defaults

-qnodbxextra: Unreferenced `typedef` declarations, `struct`, `union`, and `enum` type definitions are not included in the symbol table of the object file.

Usage

Using **-qdbxextra** may make your object and executable files larger.

Predefined macros

None.

Examples

To compile `myprogram.c` so that unreferenced `typedef`, structure, union, and enumeration declarations are included in the symbol table for use with a debugger, enter:

```
xlc myprogram.c -g -qdbxextra
```

Related information

- "-qfullpath" on page 145
- "-qlinedebug" on page 209
- "-g" on page 149
- "#pragma options" on page 356
- "-qsymtab (C only)" on page 292

-qdigraph

Category

Language element control

Pragma equivalent

#pragma options [no]digraph

Purpose

Enables recognition of digraph key combinations or keywords to represent characters not found on some keyboards.

Syntax

```
               ┌─digraph───┐
►►── -q────────┴─nodigraph─┴──────────────────────────────────────────►◄
```

Defaults

- ▷ **C** **-qdigraph** when the **extc89 | extended | extc99 | stdc99** language level is in effect. **-qnodigraph** for all other language levels.
- ▷ **C++** **-qdigraph**

Usage

A digraph is a keyword or combination of keys that lets you produce a character that is not available on all keyboards. For details on digraphs, see "Digraph characters" in the *XL C/C++ Language Reference*.

Predefined macros

__DIGRAPHS__ is predefined to 1 when **-qdigraph** is in effect; otherwise it is not defined.

Examples

To disable digraph character sequences when compiling your program, enter:

```
xlc myprogram.c -qnodigraph
```

Related information
- "-qlanglvl" on page 188
- "-qtrigraph" on page 307

-qdirectstorage

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Informs the compiler that a given compilation unit may reference write-through-enabled or cache-inhibited storage.

Syntax

```
        ┌─nodirectstorage─┐
►►── -q─┴─directstorage───┴──────────────────────────────────────►◄
```

Defaults

-qnodirectstorage

Usage

Use this option with discretion. It is intended for programmers who know how the memory and cache blocks work, and how to tune their applications for optimal performance. To ensure that your application will execute correctly on all implementations, you should assume that separate instruction and data caches exist and program your application accordingly.

-qdollar

Category

Language element control

Pragma equivalent

#pragma options [no]dollar

Purpose

Allows the dollar-sign ($) symbol to be used in the names of identifiers.

When **dollar** is in effect, the dollar symbol $ in an identifier is treated as a base character.

Syntax

```
        ┌─nodollar─┐
►►── -q─┴─dollar───┴──────────────────────────────────────────────►◄
```

Defaults

-qnodollar

Usage

If **nodollar** and the **ucs** language level are both in effect, the dollar symbol is treated as an extended character and translated into \u0024.

Predefined macros

None.

Examples

To compile `myprogram.c` so that $ is allowed in identifiers in the program, enter:

```
xlc myprogram.c -qdollar
```

Related information
• "-qlanglvl" on page 188

-qdump_class_hierarchy (C++ only)
Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Dumps a representation of the hierarchy and virtual function table layout of each class object to a file.

Syntax

```
►►── -q─dump_class_hierarchy──────────────────────────────────────────►◄
```

Defaults

Not applicable.

Usage

The output file name consists of the source file name appended with a .class suffix.

Predefined macros

None.

Examples

To compile `myprogram.C` to produce a file named `myprogram.C.class` containing the class hierarchy information, enter:

```
xlc++ myprogram.C -qdump_class_hierarchy
```

-e
Category

Linking

Pragma equivalent

None.

Purpose

When used together with the **-qmkshrobj**, specifies an entry point for a shared object.

Syntax

```
►►── -e────entry_name──────────────────────────────────────►◄
```

Defaults

None.

Parameters

name
    The name of the entry point for the shared executable.

Usage

Specify the **-e** option only with the **-qmkshrobj** option. For more information, see the description for **-qmkshrobj**.

Note: When you link object files, do not use the **-e** option. The default entry point of the executable output is __start. Changing this label with the **-e** flag can produce errors.

Predefined macros

None.

Related information
- "-qmkshrobj" on page 224

-E

Category

Output control

Pragma equivalent

None.

Purpose

Preprocesses the source files named in the compiler invocation, without compiling, and writes the output to the standard output.

Syntax

```
►►── -E─────────────────────────────────────────────────────────────►◄
```

Defaults

By the default, source files are preprocessed, compiled, and linked to produce an executable file.

Usage

The **-E** option accepts any file name. Source files with unrecognized file name suffixes are treated and preprocessed as C files, and no error message is generated.

Unless **-qnoppline** is specified, `#line` directives are generated to preserve the source coordinates of the tokens. Continuation sequences are preserved.

Unless **-C** is specified, comments are replaced in the preprocessed output by a single space character. New lines and `#line` directives are issued for comments that span multiple source lines.

The **-E** option overrides the **-P**, **-o**, and **-qsyntaxonly** options.

Predefined macros

None.

Examples

To compile `myprogram.c` and send the preprocessed source to standard output, enter:

```
xlc myprogram.c -E
```

If `myprogram.c` has a code fragment such as:

```
#define SUM(x,y) (x + y)
int a ;
#define mm 1   /* This is a comment in a
                  preprocessor directive */
int b ;        /* This is another comment across
                  two lines */
int c ;
               /* Another comment */
c = SUM(a, /* Comment in a macro function argument*/
      b) ;
```

the output will be:

```
#line 2 "myprogram.c"
int a ;
#line 5
int b ;

int c ;

c = a + b ;
```

Related information

-qeh (C++ only)

Category

Object code control

Pragma equivalent

None.

Purpose

Controls whether exception handling is enabled in the module being compiled.

When **-qeh** is in effect, exception handling is enabled. If your program does not use C++ structured exception handling, you can compile with **-qnoeh** to prevent generation of code that is not needed by your application.

Syntax

```
          ┌─eh──┐
►►── -q───┴─noeh─┴──────────────────────────────────────────────►◄
```

Defaults

-qeh

Usage

Specifying **-qeh** also implies **-qrtti**. If **-qeh** is specified together with **-qnortti**, RTTI information will still be generated as needed.

Predefined macros

__EXCEPTIONS is predefined to 1 when **-qeh** is in effect; otherwise, it is undefined.

Related information

-qenum

Category

Floating-point and integer control

Pragma equivalent

#pragma options enum, #pragma enum

Purpose

Specifies the amount of storage occupied by enumerations.

Syntax

Option syntax

```
►►── -q─enum──=──┬─intlong─┬──────────────────────────────────────────────►◄
                 ├─int─────┤
                 ├─small───┤
                 ├─1───────┤
                 ├─2───────┤
                 ├─4───────┤
                 └─8───────┘
```

Pragma syntax

```
►►──#──pragma──enum──(──┬─intlong─┬──)──────────────────────────────────►◄
                        ├─int─────┤
                        ├─small───┤
                        ├─1───────┤
                        ├─2───────┤
                        ├─4───────┤
                        ├─8───────┤
                        ├─pop─────┤
                        └─reset───┘
```
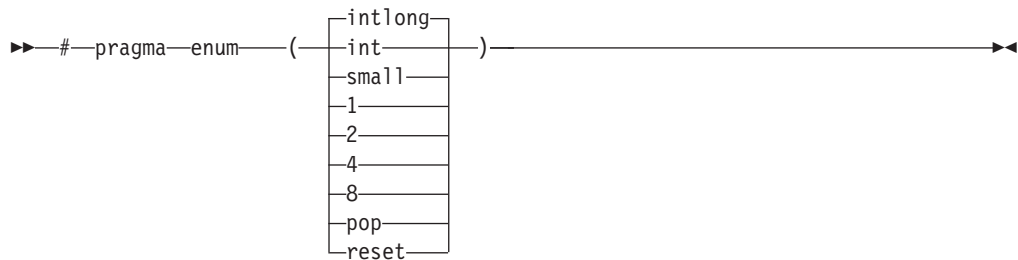
Defaults

-qenum=intlong

Parameters

1   Specifies that enumerations occupy 1 byte of storage, are of type `signed char` if the range of enumeration values falls within the limits of `signed char`, and `unsigned char` otherwise.

2   Specifies that enumerations occupy 2 bytes of storage, are of type `short` if the range of enumeration values falls within the limits of `signed short`, and `unsigned short` otherwise. ► C ◄ Values cannot exceed the range of `signed int`.

4   Specifies that enumerations occupy 4 bytes of storage, are of type `int` if the range of enumeration values falls within the limits of `signed int`, and `unsigned int` otherwise.

8   Specifies that enumerations occupy 8 bytes of storage. In 32-bit compilation mode, the enumeration is of type `long long` if the range of enumeration values falls within the limits of `signed long long`, and `unsigned long long` otherwise. In 64-bit compilation mode, the enumeration is of type `long` if the range of enumeration values falls within the limits of `signed long`, and `unsigned long` otherwise.

int

► C ◄ Specifies that enumerations occupy 4 bytes of storage and are of type `int`.

▶ C++ Specifies that enumerations occupy 4 bytes of storage, are of type `int` if the range of enumeration values falls within the limits of `signed int`, and `unsigned int` otherwise.

intlong
Specifies that enumerations occupy 8 bytes of storage, as with the **8** suboption, if the range of values in the enumeration cannot be represented by one of `int` or `unsigned int`. Otherwise, the enumerations occupy 4 bytes of storage as with the **4** suboption.

small
Specifies that enumerations occupy the smallest amount of space (1, 2, 4, or 8 bytes of storage) that can accurately represent the range of values in the enumeration. Signedness is `unsigned`, unless the range of values includes negative values. If an 8-byte `enum` results, the actual enumeration type used is dependent on compilation mode.

pop | reset (pragma only)
Discards the current pragma setting and reverts to the setting specified by the previous pragma directive. If no previous pragma was specified, reverts to the command-line or default option setting.

Usage

The tables that follow show the priority for selecting a predefined type. The table also shows the predefined type, the maximum range of `enum` constants for the corresponding predefined type, and the amount of storage that is required for that predefined type, that is, the value that the `sizeof` operator would yield when applied to the minimum-sized `enum`. All types are `signed` unless otherwise noted.

Table 23. Enumeration sizes and types

| | enum=1 | | enum=2 | | enum=4 | | enum=8 | | | |
| | | | | | | | 32-bit compilation mode | | 64-bit compilation mode | |
Range	var	const	var	const	var	const	var	const	var	const
0..127	signed char	int	short	int	int	int	long long	long long	long	long
-128..127	signed char	int	short	int	int	int	long long	long long	long	long
0..255	unsigned char	int	short	int	int	int	long long	long long	long	long
0..32767	ERROR[1]	int	short	int	int	int	long long	long long	long	long
-32768..32767	ERROR[1]	int	short	int	int	int	long long	long long	long	long
0..65535	ERROR[1]	int	unsigned short	int	int	int	long long	long long	long	long
0..2147483647	ERROR[1]	int	ERROR[1]	int	int	int	long long	long long	long	long
-(2147483647+1)..2147483647	ERROR[1]	int	ERROR[1]	int	int	int	long long	long long	long	long
0..4294967295	ERROR[1]	unsigned int[2]	ERROR[1]	unsigned int[2]	unsigned int[2]	unsigned int[2]	long long	long long	long	long
$0..(2^{63}-1)$	ERROR[1]	long[2]	ERROR[1]	long[2]	ERROR[1]	long[2]	long long[2]	long long[2]	long[2]	long[2]
$-2^{63}..(2^{63}-1)$	ERROR[1]	long[2]	ERROR[1]	long[2]	ERROR[1]	long[2]	long long[2]	long long[2]	long[2]	long[2]

Table 23. Enumeration sizes and types (continued)

0..2^64	ERROR[1]	unsigned long[2]	ERROR[1]	unsigned long[2]	ERROR[1]	unsigned long[2]	unsigned long long[2]	unsigned long long[2]	unsigned long[2]	unsigned long[2]

	enum=int		enum=intlong				enum=small			
			32-bit compilation mode		64-bit compilation mode		32-bit compilation mode		64-bit compilation mode	
Range	var	const	var	const	var	const	var	const	var	const
0..127	int	int	int	int	int	int	unsigned char	int	unsigned char	int
-128..127	int	int	int	int	int	int	signed char	int	signed char	int
0..255	int	int	int	int	int	int	unsigned char	int	unsigned char	int
0..32767	int	int	int	int	int	int	unsigned short	int	unsigned short	int
-32768..32767	int	int	int	int	int	int	short	int	short	int
0..65535	int	int	int	int	int	int	unsigned short	int	unsigned short	int
0..2147483647	int	int	int	int	int	int	unsigned int	unsigned int	unsigned int	unsigned int
-(2147483647+1)..2147483647	int	int	int	int	int	int	int	int	int	int
0..4294967295	unsigned int[1]	unsigned int[2]	unsigned int[2]	unsigned int[2]	unsigned int[2]	unsigned int[2]	unsigned int[2]	unsigned int[2]	unsigned int[2]	unsigned int[2]
0..(2^63-1)	ERR[2]	ERR[2]	long long[2]	long long[2]	long[2]	long[2]	unsigned long long[2]	unsigned long long[2]	unsigned long[2]	unsigned long[2]
-2^63..(2^63-1)	ERR[2]	ERR[2]	long long[2]	long long[2]	long[2]	long[2]	long long[2]	long long[2]	long[2]	long[2]
0..2^64	ERR[2]	ERR[2]	unsigned long long[2]	unsigned long long[2]	unsigned long[2]	unsigned long[2]	unsigned long long[2]	unsigned long long[2]	unsigned long[2]	unsigned long[2]

Note:

- These enumerations are too large for the **-qenum=1|2|4|** ▶ **C** **int** **C** ◀ setting. A Severe error is issued and compilation stops. To correct this condition, you should reduce the range of the enumerations, choose a larger **-qenum** setting, or choose a dynamic **-qenum** setting, such as **small** or **intlong**.

- ▶ **C** Enumeration types must not exceed the range of int when compiling C applications to ISO C 1989 and ISO C 1999 Standards. With the **stdc89 | stdc99** language level in effect, the compiler will behave as follows if the value of an enumeration exceeds the range of int and the **-qenum** option in effect supports this value:

  – If **-qenum=int** is in effect, a severe error message is issued and compilation stops.

  – For all other settings of **-qenum**, an informational message is issued and compilation continues.

The **#pragma enum** directive must be precede the declaration of enum variables that follow; any directives that occur within a declaration are ignored and diagnosed with a warning.

For each **#pragma enum** directive that you put in a source file, it is good practice to have a corresponding **#pragma enum=reset** before the end of that file. This should prevent one file from potentially changing the setting of another file that includes it.

Examples

If the following fragment is compiled with the **enum=small** option:

```
enum e_tag {a, b, c} e_var;
```

the range of enumeration constants is 0 through 2. This range falls within all of the ranges described in the table above. Based on priority, the compiler uses predefined type `unsigned char`.

If the following fragment is compiled with the **enum=small** option:

```
enum e_tag {a=-129, b, c} e_var;
```

the range of enumeration constants is -129 through -127. This range only falls within the ranges of `short` (`signed short`) and `int` (`signed int`). Because `short` (`signed short`) is smaller, it will be used to represent the `enum`.

The following code segment generates a warning and the second occurrence of the **enum** pragma is ignored:

```
#pragma enum=small
enum e_tag {
  a,
  b,
  #pragma enum=int /* error: cannot be within a declaration */
  c
} e_var;
#pragma enum=reset
#pragma enum=reset /* second reset isn't required */
```

Predefined macros

None.

-F

Category

Compiler customization

Pragma equivalent

None.

Purpose

Names an alternative configuration file or stanza for the compiler.

Syntax

```
►►─ -F─┬─file_path─────────────┬────────────────────────►◄
       │          └─:─stanza─┘ │
       └─:─stanza──────────────┘
```

Defaults

By default, the compiler uses the configuration file that is configured at installation time, and uses the stanza defined in that file for the invocation command currently being used.

Parameters

file_path
> The full path name of the alternate compiler configuration file to use.

stanza
> The name of the configuration file stanza to use for compilation. This directs the compiler to use the entries under that *stanza* regardless of the invocation command being used. For example, if you are compiling with **xlc**, but you specify the **c99** stanza, the compiler will use all the settings specified in the **c99** stanza.

Usage

Note that any file names or stanzas that you specify with the **-F** option override the defaults specified in the system configuration file. If you have specified a custom configuration file with the XLC_USR_CONFIG environment variable, that file is processed before the one specified by the -F option.

The **-B**, **-t**, and **-W** options override the **-F** option.

Predefined macros

None.

Examples

To compile `myprogram.c` using a stanza called `debug` that you have added to the default configuration file, enter:

```
xlc myprogram.c -F:debug
```

To compile `myprogram.c` using a configuration file called `/usr/tmp/myconfig.cfg`, enter:

```
xlc myprogram.c -F/usr/tmp/myconfig.cfg
```

To compile `myprogram.c` using the stanza `c99` you have created in a configuration file called `/usr/tmp/myconfig.cfg`, enter:

```
xlc myprogram.c -F/usr/tmp/myconfig.cfg:c99
```

Related information
- "Using custom compiler configuration files" on page 32
- "-B" on page 103
- "-t" on page 294
- "-W" on page 321
- "Specifying compiler options in a configuration file" on page 7
- "Compile-time and link-time environment variables" on page 24

-qfdpr

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Provides object files with information that the IBM Feedback Directed Program Restructuring (FDPR) performance-tuning utility needs to optimize the resulting executable file.

When **-qfdpr** is in effect, optimization data is stored in the object file.

Syntax

```
          ┌─nofdpr─┐
►►── -q────┴─fdpr──┴──────────────────────────────────────────►◄
```

Defaults

-qnofdpr

Usage

For best results, use **-qfdpr** for all object files in a program; FDPR will perform optimizations only on the files compiled with **-qfdpr**, and not library code, even if it is statically linked.

The optimizations that the FDPR utility performs are similar to those that the **-qpdf** option performs.

The FDPR performance-tuning utility has its own set of restrictions, and it is not guaranteed to speed up all programs or produce executables that produce exactly the same results as the original programs.

Predefined macros

None.

Examples

To compile myprogram.c so it includes data required by the FDPR utility, enter:

```
xlc myprogram.c -qfdpr
```

Related information

- "-qpdf1, -qpdf2" on page 236

-qflag

Category

Listings, messages, and compiler information

Pragma equivalent

#pragma options flag, "#pragma report (C++ only)" on page 368

Purpose

Limits the diagnostic messages to those of a specified severity level or higher.

The messages are written to standard output and, optionally, to the listing file if one is generated.

Syntax

-qflag syntax – C
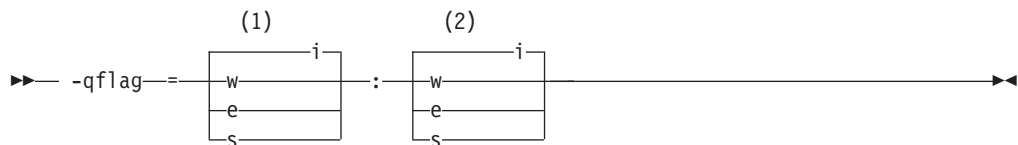
```
                         (1)                (2)
                          i                  i
  ►►── -qflag──=──┬─w─┬──────:──┬─w─┬──────────────────────────────►◄
                  ├─e─┤         ├─e─┤
                  └─s─┘         └─s─┘
```

Notes:

1   Minimum severity level of messages reported in listing

2   Minimum severity level of messages reported on terminal

-qflag syntax – C++
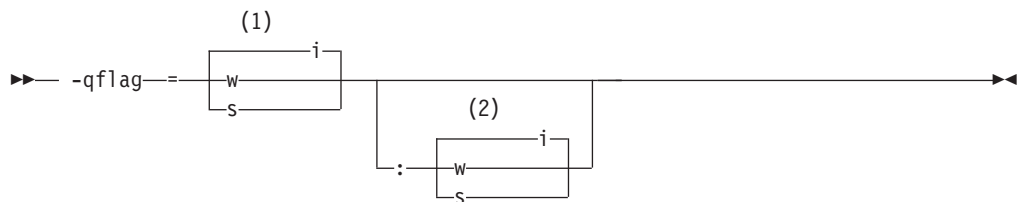
```
                       (1)
                        i
  ►►─── -qflag──=──┬─w─┬─────────────────────────────────────────►◄
                   └─s─┘              (2)
                                       i
                          └─:──┬─w─┬──────┘
                               └─s─┘
```

Notes:

1   Minimum severity level of messages reported in listing

2   Minimum severity level of messages reported on terminal

Defaults

-qflag=i : i, which shows all compiler messages

Parameters

i   Specifies that all diagnostic messages are to display: warning, error and informational messages. Informational messages (I) are of the lowest severity.

w   Specifies that warning (W) and all types of error messages are to display.

> **C**   **e**

Specifies that only error (E), severe error (S), and unrecoverable error (U) messages are to display.

s   > **C**   Specifies that only severe error (S) and unrecoverable error (U) messages are to display.   > **C++**   Specifies that only severe error (S) messages are to display.

Usage

> **C**   You must specify a minimum message severity level for both listing and terminal reporting.

> **C++**   You must specify a minimum message severity level for the listing. If you do not specify a suboption for the terminal, the compiler assumes the same severity as for the listing.

Note that using **-qflag** does not enable the classes of informational message controlled by the **-qinfo** option; see **-qinfo** for more information.

Predefined macros

None.

Examples

To compile myprogram.c so that the listing shows all messages that were generated and your workstation displays only error and higher messages (with their associated information messages to aid in fixing the errors), enter:

```
xlc myprogram.c -qflag=i:e
```

Related information
- "-qinfo" on page 164
- "-w" on page 320
- "Compiler messages" on page 16

-qfloat

Category

Floating-point and integer control

Pragma equivalent

#pragma options float

Purpose

Selects different strategies for speeding up or improving the accuracy of floating-point calculations.

Syntax

```
                                   ┌──: ◄─────────────────┐
                                   │  ┌─nospnans────────┐ │
                                   │  ├─norsqrt─────────┤ │
                                   │  ├─norrm───────────┤ │
                                   │  ├─rngchk──────────┤ │
                                   │  ├─norelax─────────┤ │
                                   │  ├─nonans──────────┤ │
                                   │  ├─maf─────────────┤ │
                                   │  ├─nohsflt─────────┤ │
                                   │  ├─nohscmplx───────┤ │
                                   │  ├─gcclongdouble───┤ │
                                   │  ├─fold────────────┤ │
                                   │  ├─nofltint────────┤ │
                                   │  ├─nofenv──────────┤ │
                                   │  ├─complexgcc──────┤ │
►►── -q─float──=──┴─────────────────┼─nocomplexgcc──────┼─┴──────────►◄
                                      ├─fenv────────────┤
                                      ├─fltint──────────┤
                                      ├─nofold──────────┤
                                      ├─nogcclongdouble─┤
                                      ├─hscmplx─────────┤
                                      ├─hsflt───────────┤
                                      ├─nomaf───────────┤
                                      ├─nans────────────┤
                                      ├─relax───────────┤
                                      ├─norngchk────────┤
                                      ├─rrm─────────────┤
                                      ├─rsqrt───────────┤
                                      └─spnans──────────┘
```
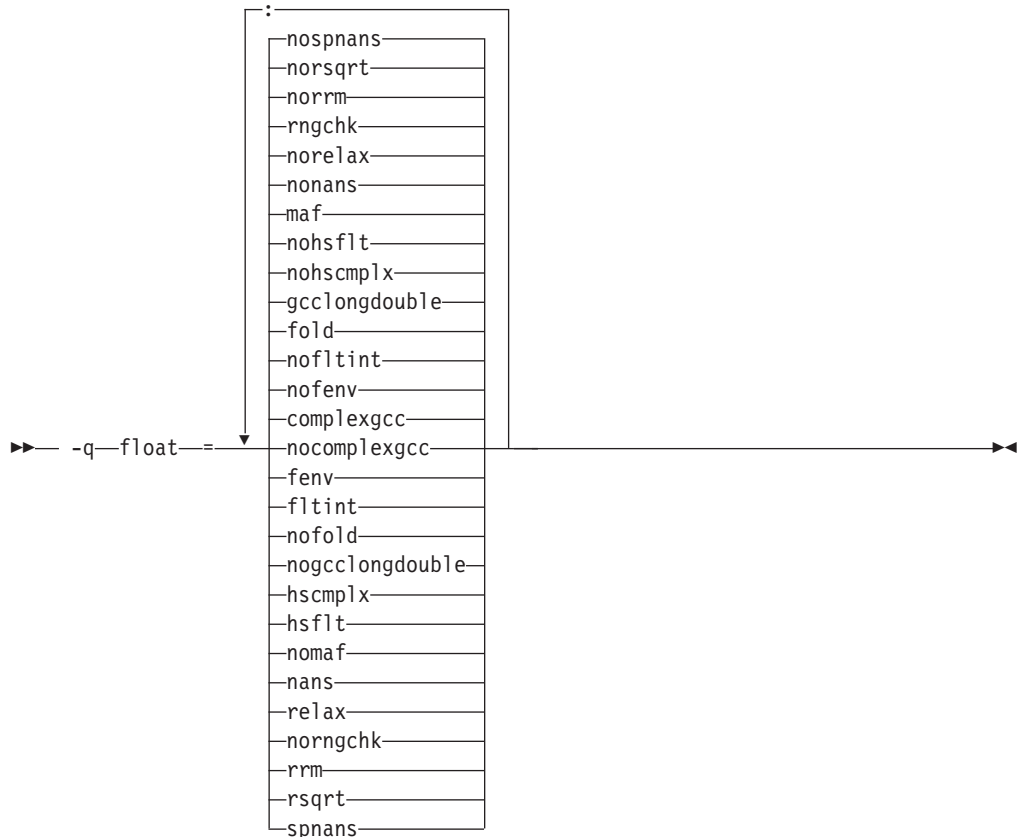
Defaults

- **-qfloat=complexgcc:nofenv:nofltint:fold:gcclongdouble:nohscmplx:nohsflt:**

  maf:nonans:norelax:rngchk:norrm:norsqrt:nospnans
- **-qfloat=fltint:rsqrt:norngchk** when **-qnostrict**,
  -qstrict=nooperationprecision:noexceptions, or **-O3** or higher optimization level
  is in effect.
- **-qfloat=nocomplexgcc** when 64-bit mode is enabled.

Parameters

complexgcc | nocomplexgcc
　　Specifies whether GCC conventions for passing or returning complex numbers
　　are to be used. **complexgcc** preserves compatibility with GCC-compiled code.
　　This suboption does not have any effect if support for complex types is not in
　　effect; see "-qlanglvl" on page 188 for details.

fenv | nofenv
　　Specifies whether the code depends on the hardware environment and whether
　　to suppress optimizations that could cause unexpected results due to this
　　dependency.

　　Certain floating-point operations rely on the status of Floating-Point Status and
　　Control Register (FPSCR), for example, to control the rounding mode or to
　　detect underflow. In particular, many compiler built-in functions read values
　　directly from the FPSCR.

　　When **nofenv** is in effect, the compiler assumes that the program does not
　　depend on the hardware environment, and that aggressive compiler

optimizations that change the sequence of floating-point operations are allowed. When **fenv** is in effect, such optimizations are suppressed.

You should use **fenv** for any code containing statements that read or set the hardware floating-point environment, to guard against optimizations that could cause unexpected behavior.

Any directives specified in the source code (such as the standard C FENV_ACCESS pragma) take precedence over the option setting.

fltint | nofltint
Speeds up floating-point-to-integer conversions by using an inline sequence of code instead of a call to a library function. The library function, which is called when **nofltint** is in effect, checks for floating-point values outside the representable range of integers and returns the minimum or maximum representable integer if passed an out-of-range floating-point value.

If **-qarch** is set to a processor that has an instruction to convert from floating point to integer, that instruction will be used regardless of the **[no]fltint** setting. This conversion also applies to all Power processors in 64-bit mode.

If you compile with **-O3** or higher optimization level, **fltint** is enabled automatically. To disable it, also specify **-qstrict**, **-qstrict=operationprecision**, or **-qstrict=exceptions**.

fold | nofold
Evaluates constant floating-point expressions at compile time, which may yield slightly different results from evaluating them at run time. The compiler always evaluates constant expressions in specification statements, even if you specify **nofold**.

gcclongdouble | nogcclongdouble
Specifies whether the compiler uses GCC-supplied or IBM-supplied library functions for 128-bit long double operations.

gcclongdouble ensures binary compatibility with GCC for mathematical calculations. If this compatibility is not important in your application, you should use **nogcclongdouble** for better performance. This suboption only has an effect when 128-bit long double types are enabled with **-qldbl128**.

Note: Passing results from modules compiled with **nogcclongdouble** to modules compiled with **gcclongdouble** may produce different results for numbers such as Inf, NaN and other rare cases. To avoid such incompatibilities, the compiler provides built-in functions to convert IBM long double types to GCC long double types; see "Binary floating-point built-in functions" on page 414 for more information.

hscmplx | nohscmplx
Speeds up operations involving complex division and complex absolute value. This suboption, which provides a subset of the optimizations of the **hsflt** suboption, is preferred for complex calculations.

hsflt | nohsflt
Speeds up calculations by preventing rounding for single-precision expressions and by replacing floating-point division by multiplication with the reciprocal of the divisor. It also uses the same technique as the **fltint** suboption for floating-point-to-integer conversions. **hsflt** implies **hscmplx**.

The **hsflt** suboption overrides the **nans** and **spnans** suboptions.

Note: Use **-qfloat=hsflt** on applications that perform complex division and floating-point conversions where floating-point calculations have known characteristics. In particular, all floating-point results must be within the defined range of representation of single precision. Use with discretion, as this option may produce unexpected results without warning. For complex computations, it is recommended that you use the **hscmplx** suboption (described above), which provides equivalent speed-up without the undesirable results of **hsflt**.

maf | nomaf
Makes floating-point calculations faster and more accurate by using floating-point multiply-add instructions where appropriate. The results may not be exactly equivalent to those from similar calculations performed at compile time or on other types of computers. Negative zero results may be produced. This suboption may affect the precision of floating-point intermediate results. If **-qfloat=nomaf** is specified, no multiply-add instructions will be generated unless they are required for correctness.

nans | nonans
Allows you to use the **-qflttrap=invalid:enable** option to detect and deal with exception conditions that involve signaling NaN (not-a-number) values. Use this suboption only if your program explicitly creates signaling NaN values, because these values never result from other floating-point operations.

relax | norelax
Relaxes strict IEEE conformance slightly for greater speed, typically by removing some trivial floating-point arithmetic operations, such as adds and subtracts involving a zero on the right. These changes are allowed if either **-qstrict=noieeefp** or **-qfloat=relax** is specified.

rngchk | norngchk
At optimization level **-O3** and above, and without **-qstrict**, controls whether range checking is performed for input arguments for software divide and inlined square root operations. Specifying **norngchk** instructs the compiler to skip range checking, allowing for increased performance where division and square root operations are performed repeatedly within a loop.

Note that with **norngchk** in effect the following restrictions apply:
- The dividend of a division operation must not be +/-INF.
- The divisor of a division operation must not be 0.0, +/- INF, or denormalized values.
- The quotient of dividend and divisor must not be +/-INF.
- The input for a square root operation must not be INF.

If any of these conditions are not met, incorrect results may be produced. For example, if the divisor for a division operation is 0.0 or a denormalized number (absolute value $< 2^{-1022}$ for double precision, and absolute value $< 2^{-126}$ for single precision), NaN, instead of INF, may result; when the divisor is +/- INF, NaN instead of 0.0 may result. If the input is +INF for a sqrt operation, NaN, rather than INF, may result.

norngchk is only allowed when **-qnostrict** is in effect. If **-qstrict**, **-qstrict=infinities**, **-qstrict=operationprecision**, or **-qstrict=exceptions** is in effect, **norngchk** is ignored.

rrm | norrm
Prevents floating-point optimizations that require the rounding mode to be the default, round-to-nearest, at run time, by informing the compiler that the floating-point rounding mode may change or is not round-to-nearest at run

time. You should use **rrm** if your program changes the runtime rounding mode by any means; otherwise, the program may compute incorrect results.

rsqrt | <u>**norsqrt**</u>
Speeds up some calculations by replacing division by the result of a square root with multiplication by the reciprocal of the square root.

rsqrt has no effect unless **-qignerrno** is also specified; errno will *not* be set for any sqrt function calls.

If you compile with **-O3** or higher optimization level, **rsqrt** is enabled automatically. To disable it, also specify **-qstrict**, **-qstrict=nans**, **-qstrict=infinities**, **-qstrict=zerosigns**, or **-qstrict=exceptions**.

spnans | <u>**nospnans**</u>
Generates extra instructions to detect signalling NaN on conversion from single-precision to double-precision.

Note:

- For details about the relationship between **-qfloat** suboptions and their **-qstrict** counterparts, see "-qstrict" on page 286.

Usage

Using **-qfloat** suboptions other than the default settings may produce incorrect results in floating-point computations if not all required conditions for a given suboption are met. For this reason, you should only use this option if you are experienced with floating-point calculations involving IEEE floating-point values and can properly assess the possibility of introducing errors in your program. See also "Handling floating point operations" in the *XL C/C++ Optimization and Programming Guide* for more information.

If the **-qstrict** | **-qnostrict** and **float** suboptions conflict, the last setting specified is used.

Predefined macros

Examples

To compile myprogram.c so that constant floating point expressions are evaluated at compile time and multiply-add instructions are not generated, enter:

```
xlc myprogram.c -qfloat=fold:nomaf
```

Related information
- "-qarch" on page 95
- "-qcomplexgccincl" on page 115
- "-qflttrap"
- "-qldbl128" on page 206
- "-qstrict" on page 286

-qflttrap
Category

Error checking and debugging

Pragma equivalent

#pragma options [no]flttrap

Purpose

Determines the types of floating-point exception conditions to be detected at run time

Syntax

```
►►── -q ──┬─noflttrap──────────────────────────────────────┬──────────────────►◄
          └─flttrap──┬───────────────────────────────────┬─┘
                     │          ┌─ : ─────────────┐        │
                     │          │  ┌─zero──────┐  │        │
                     │          │  ├─zerodivide─┤  │        │
                     │          │  ├─und───────┤  │        │
                     │          │  ├─underflow─┤  │        │
                     │          │  ├─ov────────┤  │        │
                     │          │  ├─overflow──┤  │        │
                     │          │  ├─inv───────┤  │        │
                     │          │  ├─invalid───┤  │        │
                     │          │  ├─inex──────┤  │        │
                     │          │  ▼  ├─inexact───┤  │      │
                     └─ = ──────┴──┼─enable────┼──┘        │
                                   ├─en────────┤           │
                                   └─nanq──────┘           │
```

Defaults

-qnoflttrap

Parameters

enable, en
> Enables trapping when the specified exceptions (**overflow**, **underflow**, **zerodivide**, **invalid**, or **inexact**) occur. You must specify this suboption if you want to turn on exception trapping without modifying your source code. If any of the specified exceptions occur, a SIGTRAP or SIGFPE signal is sent to the process with the precise location of the exception.

inexact, inex
> Enables the detection of floating-point inexact operations. If a floating-point inexact operation occurs, an inexact operation exception status flag is set in the Floating-Point Status and Control Register (FPSCR).

invalid, inv
> Enables the detection of floating-point invalid operations. If a floating-point invalid operation occurs, an invalid operation exception status flag is set in the FPSCR.

nanq
> Generates code to detect NaNQ (Not a Number Quiet) and NaNS (Not a Number Signalling) exceptions before and after each floating point operation, including assignment, and after each call to a function returning a floating-point result to trap if the value is a NaN. Trapping code is generated regardless of whether the **enable** suboption is specified.

overflow, ov

Enables the detection of floating-point overflow. If a floating-point overflow occurs, an overflow exception status flag is set in the FPSCR.

underflow, und

Enables the detection of floating-point underflow. If a floating-point underflow occurs, an underflow exception status flag is set in the FPSCR.

zerodivide, zero

Enables the detection of floating-point division by zero. If a floating-point zero-divide occurs, a zero-divide exception status flag is set in the FPSCR.

Specifying **-qflttrap** option with no suboptions is equivalent to **-qflttrap=overflow : underflow : zerodivide : invalid : inexact**. Exceptions will be detected by the hardware, but trapping is not enabled. Because this default does not include **enable**, it is probably only useful if you already use `fpsets` or similar subroutines in your source.

Usage

It is recommended that you use the **enable** suboption whenever compiling the `main` program with **-qflttrap**. This ensures that the compiler will generate the code to automatically enable floating-point exception trapping, without requiring that you include calls to the appropriate floating-point exception library functions in your code.

If you specify **-qflttrap** more than once, both with and without suboptions, the **-qflttrap** without suboptions is ignored.

This option is recognized during linking with IPA. Specifying the option at the link step overrides the compile-time setting.

If your program contains signalling NaNs, you should use the **-qfloat=nans** option along with **-qflttrap** to trap any exceptions.

The compiler exhibits behavior as illustrated in the following examples when the **-qflttrap** option is specified together with an optimization option:
* with **-O2**:
  – 1/0 generates a **div0** exception and has a result of infinity
  – 0/0 generates an invalid operation
* with **-O3** or greater:
  – 1/0 generates a **div0** exception and has a result of infinity
  – 0/0 returns zero multiplied by the result of the previous division.

Note: Due to the transformations performed and the exception handling support of some vector instructions, use of **-qsimd=auto** may change the location where an exception is caught or even cause the compiler to miss catching an exception.

Predefined macros

None.

Examples

When you compile this program:

```
#include <stdio.h>

int main()
{
  float x, y, z;
  x = 5.0;
  y = 0.0;
  z = x / y;
  printf("%f", z);
}
```

with the command:

```
xlc -qflttrap=zerodivide:enable divide_by_zero.c
```

the program stops when the division is performed.

The **zerodivide** suboption identifies the type of exception to guard against. The
enable suboption causes a SIGTRAP or SIGFPE signal to be generated when the
exception occurs.

Related information
- "-qfloat" on page 137
- "-qarch" on page 95

-qformat

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Warns of possible problems with string input and output format specifications.

Functions diagnosed are `printf`, `scanf`, `strftime`, `strfmon` family functions and
functions marked with format attributes.

Syntax
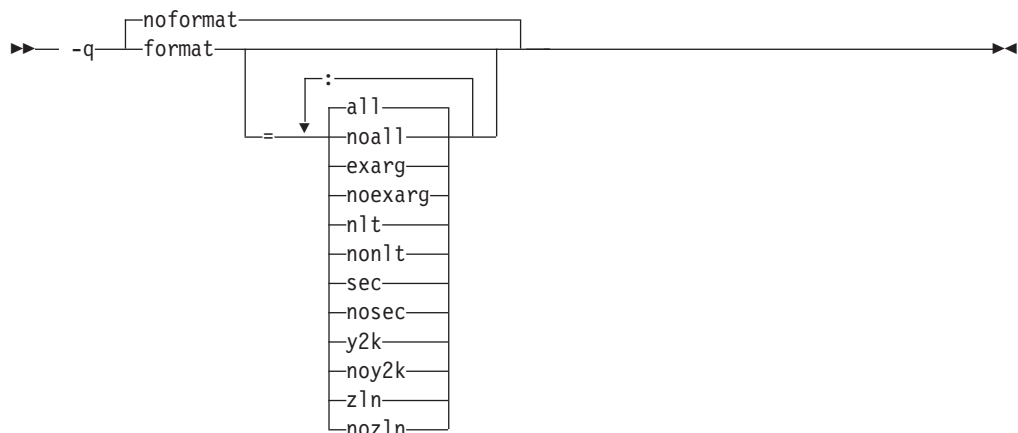
Defaults

-qnoformat

Parameters

all | **noall**
Enables or disables all format diagnostic messages.

exarg | **noexarg**
Warns if excess arguments appear in `printf` and `scanf` style function calls.

nlt | **nonlt**
Warns if a format string is not a string literal, unless the format function takes its format arguments as a `va_list`.

sec | **nosec**
Warns of possible security problems in use of format functions.

y2k | **noy2k**
Warns of `strftime` formats that produce a 2-digit year.

zln | **nozln**
Warns of zero-length formats.

Specifying **-qformat** with no suboptions is equivalent to **-qformat=all**.

-qnoformat is equivalent to **-qformat=noall**.

Predefined macros

None.

Examples

To enable all format string diagnostics, enter either of the following:

```
xlc myprogram.c -qformat=all
xlc myprogram.c -qformat
```

To enable all format diagnostic checking except that for y2k date diagnostics, enter:

```
xlc myprogram.c -qformat=all:noy2k
```

-qfullpath

Category

Error checking and debugging

Pragma equivalent

#pragma options [no]fullpath

Purpose

When used with the **-g** or **-qlinedebug** option, this option records the full, or absolute, path names of source and include files in object files compiled with debugging information, so that debugging tools can correctly locate the source files.

When **fullpath** is in effect, the absolute (full) path names of source files are preserved. When **nofullpath** is in effect, the relative path names of source files are preserved.

Syntax

```
►►── -q───┬─nofullpath─┬──────────────────────────────────────────────────◄◄
          └─fullpath───┘
```

Defaults

-qnofullpath

Usage

If your executable file was moved to another directory, the debugger would be unable to find the file unless you provide a search path in the debugger. You can use **fullpath** to ensure that the debugger locates the file successfully.

Predefined macros

None.

Related information
- "-qlinedebug" on page 209
- "-g" on page 149

-qfunctrace
Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Calls to the tracing routine that traces the entry and exit points of functions in a compilation unit, or only a specific list of functions.

Syntax

```
       ┌─-qnofunctrace──────────────────────────────────────────────┐
►►──┼─-qfunctrace──┬───────────────────────────────────────────┬──┼──◄◄
                   │                      ┌──:──┐               │
                   │      ┌─ + ─┐    ▼    ┌─function_name─┐      │
                   └──────┤     ├─────────┼─classname─────┼──────┘
                          └─ - ─┘         └─namespace─────┘
```
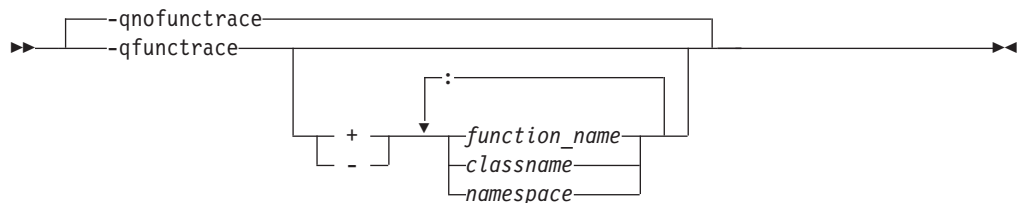
Pragma syntax

```
►►—#—pragma—nofunctrace—(——function_name——)————————►◄
                          └←———,———┘
```

Defaults

-qnofunctrace

-qfunctrace-std

Parameters

+ Instructs the compiler to trace *function_name*, *classes*, or *namespace*, and all its internal functions.

- Instructs the compiler not to trace *function_name*, *classes*, or *namespace*, or any of its internal functions.

function_name
Indicates the named functions to be traced.

classname
Indicates the named class to be traced.

namespace
Indicates the namespace to be traced.

Usage

-qfunctrace enables tracing for all functions in your program. **-qnofunctrace** disables tracing that was enabled by **-qfunctrace**.

The **-qfunctrace+** and **-qfunctrace-** suboptions enable tracing for a specific list of functions and are not affected by **-qnofunctrace**. The list of functions is cumulative.

This option inserts calls to user-defined tracing functions. These functions must be provided at the link step. For details about the interface of tracing functions, as well as when they are called, see the Tracing functions in your code section in the *XL C/C++ Optimization and Programming Guide*.

Use **+** or **-** to indicate the function , classname, or namespace to be traced by the compiler. For example, if you want to trace function x, use `-qfunctrace+x`. To trace a list of functions, you must use a colon `:` to separate them.

Two colons in a row `::` is a scope qualifier, you can use it to indicate C++ qualified names. For example, use `-qfunctrace+A::B:C` traces functions that begin with qualifiers `A::B` or `C`.

If you want to trace functions in your code, you can write tracing functions in your code by using the following C function prototypes:

* Use `void __func_trace_enter(const char *const function_name, const char *const file_name, int line_number, void **const user_data);` to define the entry point tracing routine.

- Use void __func_trace_exit(const char *const function_name, const char *const file_name, int line_number, void **const user_data); to define the exit point tracing routine.
- Use void __func_trace_catch(const char *const function_name, const char *const file_name, int line_number, void **const user_data); to define the catch tracing routine.

You must define your functions when you write the preceding function prototypes in your code.

For details about the these function prototypes as well as when they are called, see the **Tracing functions in your code** section in the *XL C/C++ Optimization and Programming Guide*.

Note:
- You can only use **+** and **-** one at a time. Do not use both of them together in the same **-qfunctrace** invocation.
- Definition of an inline function is traced. It is only the calls that have been inlined are not traced.

Predefined macros

None.

Examples

To trace functions x, y, and z, use -qfunctrace+x:y:z.

To trace all functions, except for x, use -qfunctrace -qfunctrace-x.

The **-qfunctrace+** and **-qfunctrace-** suboptions only enable or disable tracing on the given list of cumulative functions. When functions, classes, and namespaces are used, the most completely specified option is in effect. The following is a list of examples:
- -qfunctrace+x -qfunctrace+y or -qfunctrace+x -qnofunctrace -qfunctrace+y enables tracing for only x and y.
- -qfunctrace-x -qfunctrace or -qfunctrace -qfunctrace-x traces all functions in the compilation unit except for x.
- -qfunctrace -qfunctrace+x traces all functions.
- -qfunctrace+y -qnofunctrace traces y only.
- -qfunctrace-functionX -qfunctrace+classX or -qfunctrace+classX -qfunctrace-functionX (functionX is a member function of classX) traces all member functions of classX but not functionX. This is because in this example the most completely specified option wins regardless of the order you specify the option.
- -qfunctrace+MyClass traces all member functions in MyClass.
- -qfunctrace+std::vector traces all instantiations of std::vector.
- -qfunctrace+ABC -qfunctrace-ABC::foo traces all functions defined in namespace ABC except for foo.

Related information
- For details about **#pragma nofunctrace**, see "#pragma nofunctrace" on page 355.

- For detailed information about how to implement function tracing routines in your code, as well as detailed examples and a list of rules for using them, see **Tracing functions in your code** in the *XL C/C++ Optimization and Programming Guide*.

-g

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Generates debug information for use by a symbolic debugger.

Syntax

►►── -g ──►◄

Defaults

Not applicable.

Usage

Specifying **-g** will turn off all inlining unless you explicitly request it with an optimization option.

To specify that source files used with **-g** are referred to by either their absolute or their relative path name, use the **-qfullpath** option.

You can also use the **-qlinedebug** option to produce abbreviated debugging information in a smaller object size.

Predefined macros

None.

Examples

To compile `myprogram.c` to produce an executable program `testing` so you can debug it, enter:

```
xlc myprogram.c -o testing -g
```

Related information
- "-qfullpath" on page 145
- -qinline
- "-qlinedebug" on page 209
- "-O, -qoptimize" on page 227

-qgcc_c_stdinc (C only)

Category

Compiler customization

Pragma equivalent

None.

Purpose

Changes the standard search location for the GNU C system header files.

Syntax

```
►►── -q──gcc_c_stdinc──=──┬──┬──────┬──┬──▼─directory_path─┬──┬──────┬──┬──────────────►◄
                          └──┘  "  └──┘   └──── : ────────┘  └──┘  "  └──┘
```

Defaults

By default, the compiler searches the directory specified in the configuration file.

Parameters

directory_path
> The path for the directory where the compiler should search for the GNU C header files. You can surround the path with quotation marks to ensure it is not split up by the command line.

Usage

This option allows you to change the search paths for specific compilations. To permanently change the default search paths for the GNU C headers, you use a configuration file to do so; see "Directory search sequence for include files" on page 12 for more information.

If this option is specified more than once, only the last instance of the option is used by the compiler.

This option is ignored if the **-qnostdinc** option is in effect.

Predefined macros

None.

Examples

To override the default search paths for the GNU C headers with `mypath/headers1` and `mypath/headers2`, enter:

```
xlc myprogram.c -qgcc_c_stdinc=mypath/headers1:mypath:headers2
```

Related information
- "-qc_stdinc (C only)" on page 118
- "-qstdinc" on page 285
- "-qinclude" on page 162
- "Directory search sequence for include files" on page 12
- "Specifying compiler options in a configuration file" on page 7

-qgcc_cpp_stdinc (C++ only)
Category

Compiler customization

Pragma equivalent

None

Purpose

Changes the standard search location for the GNU C++ system header files.

Syntax

Defaults

By default, the compiler searches the directory specified in the configuration file.

Parameters

directory_path
> The path for the directory where the compiler should search for the GNU C++ header files. You can surround the path with quotation marks to ensure it is not split up by the command line.

Usage

This option allows you to change the search paths for specific compilations. To permanently change the default search paths for the GNU C++ headers, you use a configuration file to do so; see "Directory search sequence for include files" on page 12 for more information.

If this option is specified more than once, only the last instance of the option is used by the compiler.

This option is ignored if the **-qnostdinc** option is in effect.

Predefined macros

None.

Examples

To override the default search paths for the GNU C++ headers with
mypath/headers1 and mypath/headers2, enter:

```
xlc++ myprogram.C -qgcc_cpp_stdinc=mypath/headers1:mypath:headers2
```

Related information
- "-qcpp_stdinc (C++ only)" on page 119
- "-qstdinc" on page 285
- "-qinclude" on page 162
- "Directory search sequence for include files" on page 12
- "Specifying compiler options in a configuration file" on page 7

-qgenproto (C only)
Category

Portability and migration

Pragma equivalent

None.

Purpose

Produces prototype declarations from K&R function definitions or function
definitions with empty parentheses, and displays them to standard output.

The compiler accepts and compiles K&R function definitions or definitions with a
function declarator with empty parentheses; however, these function definitions are
considered by the C standard to be obsolete (the compiler will diagnose them if
you enable the **-qinfo=obs** option). When **-qgenproto** is in effect, the compiler
generates the corresponding prototype declarations and displays them to standard
output. You can use this option to help you identify obsolete function definitions
and automatically obtain equivalent prototypes.

Syntax

```
        ┌─nogenproto─┐
►►── -q──┴─genproto───┴──────────────────────────►◄
              └──=──parmnames──┘
```

Defaults

-qnogenproto

Parameters

parmnames
    Parameter names are included in the prototype. If you do not specify this
    suboption, parameter names will not be included in the prototype.

Predefined macros

None.

Examples

Compiling with **-** **qgenproto** for the following function definitions:

```
int foo(a, b)    // K&R function
  int a, b;
{
}

int faa(int i) { } // prototyped function

main() {  // missing void parameter
}
```

produces the following output on the display:

```
int foo(int, int);
int main(void);
```

Specifying **-qgenproto=parmnames** produces:

```
int foo(int a, int b);
int main(void);
```

-qhalt

Category

Error checking and debugging

Pragma equivalent

#pragma options halt

Purpose

Stops compilation before producing any object, executable, or assembler source files if the maximum severity of compile-time messages equals or exceeds the severity you specify.

Syntax

-qhalt syntax — C

```
►►── -qhalt──=──┬─s─┬──────────────────────────────────────────►◄
                ├─i─┤
                ├─w─┤
                └─e─┘
```

-qhalt syntax — C++

```
►►── -qhalt──=──┬─s─┬──────────────────────────────────────────►◄
                ├─i─┤
                └─w─┘
```

Defaults

-qhalt=s

Parameters

i   Specifies that compilation is to stop for all types of errors: warning, error and informational. Informational diagnostics (I) are of the lowest severity.

w   Specifies that compilation is to stop for warnings (W) and all types of errors.

> **C** **e**
> Specifies that compilation is to stop for errors (E), severe errors (S), and unrecoverable errors (U).

s̲   > **C**   Specifies that compilation is to stop for severe errors (S) and unrecoverable errors (U). ▶ **C++** Specifies that compilation is to stop for severe errors (S).

Usage

When the compiler stops as a result of the **halt** option, the compiler return code is nonzero. For a list of return codes, see "Compiler return codes" on page 18.

When **-qhalt** is specified more than once, the lowest severity level is used.

Diagnostic messages may be controlled by the **-qflag** option.

You can also instruct the compiler to stop compilation based on the number of errors of a type of severity by using the **-qmaxerr** option, which overrides **-qhalt**.

▶ **C++** You can also use the **-qhaltonmsg** option to stop compilation according to error message number.

Predefined macros

None.

Examples

To compile `myprogram.c` so that compilation stops if a warning or higher level message occurs, enter:

```
xlc myprogram.c -qhalt=w
```

Related information
- "-qhaltonmsg (C++ only)"
- "-qflag" on page 136
- "-qmaxerr" on page 219

-qhaltonmsg (C++ only)
Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Stops compilation before producing any object, executable, or assembler source files if a specified error message is generated.

Syntax

```
                              ┌─ : ──────────────────┐
                              │                      │
►►── -qhaltonmsg ──= ──▼── message_identifier ──┴─────────────────────────►◄
```

Defaults

Not applicable.

Parameters

message_identifier
> Represents a message identifier. The message identifier must be in the following format:

> 15*dd-number*

> where:

> *dd*   Is the two-digit code representing the compiler component that produces the message. See "Compiler message format" on page 16 for descriptions of these.

> *number*
> > Is the message number.

Usage

When the compiler stops as a result of the **-qhaltonmsg** option, the compiler return code is nonzero.

Predefined macros

None.

Related information
* "Compiler messages" on page 16

-qhot

Category

Optimization and tuning

Pragma equivalent

#pragma novector

Purpose

Performs high-order loop analysis and transformations (HOT) during optimization.

The **-qhot** compiler option is a powerful alternative to hand tuning that provides opportunities to optimize loops and array language. This compiler option will always attempt to optimize loops, regardless of the suboptions you specify.

You can use the pragma directives to disable these transformations for selected sections of code.

Syntax
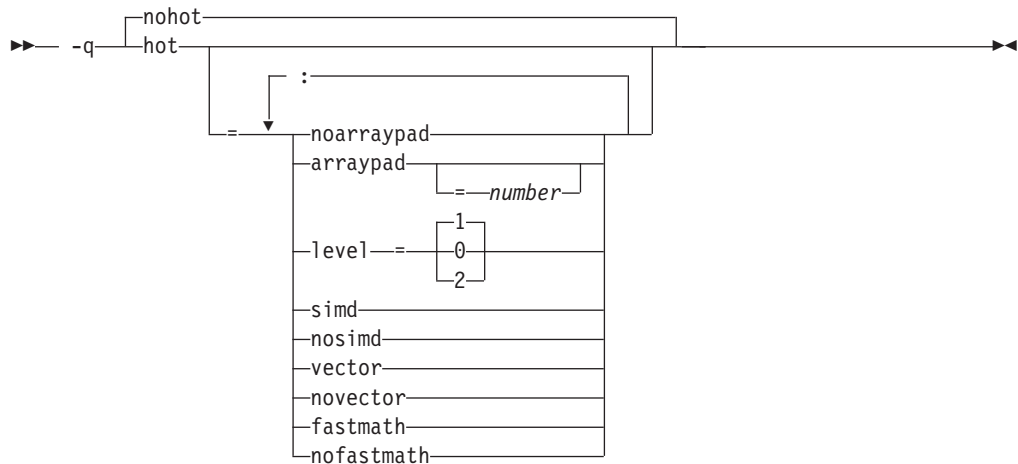
Option syntax

Pragma syntax
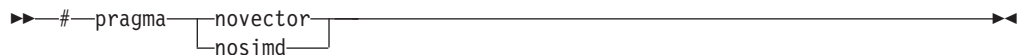
Defaults

- **-qnohot**
- **-qhot=noarraypad:level=0:novector:fastmath** when **-O3** is in effect.
- **-qhot=noarraypad:level=1:vector:fastmath** when **-qsmp**, **-O4** or **-O5** is in effect.
- Specifying **-qhot** without suboptions is equivalent to **-qhot=noarraypad:level=1:vector:fastmath**.

Parameters

arraypad | noarraypad (option only)
    Permits the compiler to increase the dimensions of arrays where doing so might improve the efficiency of array-processing loops. (Because of the implementation of the cache architecture, array dimensions that are powers of two can lead to decreased cache utilization.) Specifying **-qhot=arraypad** when your source includes large arrays with dimensions that are powers of 2 can reduce cache misses and page faults that slow your array processing programs. This can be particularly effective when the first dimension is a power of 2. If you use this suboption with no *number*, the compiler will pad any arrays where it infers there may be a benefit and will pad by whatever amount it chooses. Not all arrays will necessarily be padded, and different arrays may be padded by different amounts. If you specify a *number*, the compiler will pad every array in the code.

Note: Using **arraypad** can be unsafe, as it does not perform any checking for reshaping or equivalences that may cause the code to break if padding takes place.

number **(option only)**
A positive integer value representing the number of elements by which each array will be padded in the source. The pad amount must be a positive integer value. It is recommended that pad values be multiples of the largest array element size, typically 4, 8, or 16.

level=0 (option only)
Performs a subset of the high-order transformations and sets the default to **novector:noarraypad:fastmath**.

level=1 (option only)
Performs the default set of high-order transformations.

level=2 (option only)
Performs the default set of high-order transformations and some more aggressive loop transformations. **-qhot=level=2** must be used with **-qsmp**. This option performs aggressive loop analysis and transformations to improve cache reuse and exploit loop parallelization opportunities.

simd (option only) | nosimd
This suboption has been deprecated. Consider using the **-qsimd** compiler option.

vector (option only) | novector
When specified with **-qnostrict** and **-qignerrno**, or an optimization level of **-O3** or higher, **vector** causes the compiler to convert certain operations that are performed in a loop on successive elements of an array (for example, square root, reciprocal square root) into a call to a routine in the Mathematical Acceleration Subsystem (MASS) library in libxlopt. The **vector** suboption supports single and double-precision floating-point mathematics, and is useful for applications with significant mathematical processing demands.

novector disables the conversion of loop array operations into calls to MASS library routines.

Since vectorization can affect the precision of your program's results, if you are using **-O4** or higher, you should specify **-qhot=novector** if the change in precision is unacceptable to you.

fastmath | nofastmath
You can use this suboption to tune your application to either use fast scalar versions of math functions or use the default versions.

For C/C++, you must use this suboption together with **-qignerrno**, unless **-qignerrno** is already enabled by other options.

-qhot=fastmath enables the replacement of math routines with available math routines from the XLOPT library only if **-qstrict=nolibrary** is enabled. **-qhot=nofastmath** disables the replacement of math routines by the XLOPT library. **-qhot=fastmath** is enabled by default if **-qhot** is specified regardless of the hot level.

Usage

If you do not also specify an optimization level when specifying **-qhot** on the command line, the compiler assumes **-O2**.

If you want to override the default **level** setting of **1** when using **-qsmp**, **-O4** or **-O5**, be sure to specify **-qhot=level=0** or **-qhot=level=2** *after* the other options.

The pragma directives apply only to `while`, `do while`, and `for` loops that immediately follow the placement of the directives. They have no effect on other loops that may be nested within the specified loop.

You can use the **-qreport** option in conjunction with **-qhot** or any optimization option that implies **-qhot** to produce a pseudo-C report showing how the loops were transformed. The loop transformations are included in the listing report if either the option **-qreport** or **-qlistfmt** is also specified. This `LOOP TRANSFORMATION SECTION` of the listing file also contains information about data prefetch insertion locations. In addition, when you use `-qprefetch=assistthread` to generate prefetching assist threads, a message `Assist thread for data prefetching was generated` also appears in the `LOOP TRANSFORMATION SECTION` of the listing file. Specifying `-qprefetch=assitthread` guides the compiler to generate aggressive data prefetching at optimization level **-O3 -qhot** or higher. For more information, see "-qreport" on page 254.

Predefined macros

None.

Related information
- "-qarch" on page 95
- "-qsimd" on page 269
- "-qprefetch" on page 245
- "-qreport" on page 254
- "-O, -qoptimize" on page 227
- "-qstrict" on page 286
- "-qsmp" on page 272
- *Using the Mathematical Acceleration Subsystem (MASS)* in the *XL C/C++ Optimization and Programming Guide*

-I

Category

Input control

Pragma equivalent

None.

Purpose

Adds a directory to the search path for include files.

Syntax

▶▶── -I─*directory_path*───◀◀

Defaults

See "Directory search sequence for include files" on page 12 for a description of the default search paths.

Parameters

directory_path
> The path for the directory where the compiler should search for the header files.

Usage

If **-qnostdinc** is in effect, the compiler searches *only* the paths specified by the **-I** option for header files, and not the standard search paths as well. If **-qidirfirst** is in effect, the directories specified by the **-I** option are searched before any other directories.

If the **-I** directory option is specified both in the configuration file and on the command line, the paths specified in the configuration file are searched first. The **-I** directory option can be specified more than once on the command line. If you specify more than one **-I** option, directories are searched in the order that they appear on the command line.

The **-I** option has no effect on files that are included using an absolute path name.

Predefined macros

None.

Examples

To compile myprogram.c and search /usr/tmp and then /oldstuff/history for included files, enter:

```
xlc myprogram.c -I/usr/tmp -I/oldstuff/history
```

Related information
- "-qidirfirst"
- "-qstdinc" on page 285
- "-qinclude" on page 162
- "Directory search sequence for include files" on page 12
- "Specifying compiler options in a configuration file" on page 7

-qidirfirst

Category

Input control

Pragma equivalent

#pragma options [no]idirfirst

Purpose

Specifies whether the compiler searches for user include files in directories specified by the **-I** option *before* or *after* searching any other directories.

When **-qidirfirst** is in effect, the compiler first searches the directories specified by the **-I** option before searching any other directories. When **-qnoidirfirst** is in effect, before searching directories named on the **-I** option, the compiler first searches a) the directories in which source files named on the **-qinclude** option are located; and b) the directories in which the including files are located.

Syntax

```
>>-- -q----noidirfirst----------------------------------------------><
           '-idirfirst--'
```

Defaults

-qnoidirfirst

Usage

This option only affects files included with the `#include "file_name"` directive or the **-qinclude** option; **-qidirfirst** is independent of the **-qnostdinc** option and has no effect on the search order for XL C/C++ or system header files. (For the search order of header files, see "Directory search sequence for include files" on page 12.) This option also has no effect on files that are included using an absolute path name.

The last valid pragma directive remains in effect until replaced by a subsequent pragma.

Predefined macros

None.

Examples

To compile `myprogram.c` and search `/usr/tmp/myinclude` for included files before searching the current directory (where the source file resides), enter:

```
xlc myprogram.c -I/usr/tmp/myinclude -qidirfirst
```

Related information
- "-I" on page 158
- "-qinclude" on page 162
- "-qstdinc" on page 285
- "-qc_stdinc (C only)" on page 118
- "-qcpp_stdinc (C++ only)" on page 119
- "Directory search sequence for include files" on page 12

-qignerrno

Category

Optimization and tuning

Pragma equivalent

#pragma options [no]ignerrno

Purpose

Allows the compiler to perform optimizations that assume `errno` is not modified by system calls.

Some system library functions set `errno` when an exception occurs. When **ignerrno** is in effect, the setting and subsequent side effects of `errno` are ignored. This allows the compiler to perform optimizations that assume `errno` is not modified by system calls.

Syntax

```
►►── -q──┬─noignerrno─┬──────────────────────────────────────────►◄
         └─ignerrno───┘
```

Defaults

- -qnoignerrno
- **-qignerrno** when **-O3** or higher optimization is in effect.

Usage

If you require both **-O3** or higher and the ability to set `errno`, you should specify **-qnoignerrno** *after* the optimization option on the command line.

Predefined macros

▶ `C++`   __IGNERRNO__ is defined to 1 when **ignerrno** is in effect; otherwise, it is undefined.

Related information

- "-O, -qoptimize" on page 227

-qignprag

Category

Language element control

Pragma equivalent

#pragma options [no]ignprag

Purpose

Instructs the compiler to ignore certain pragma statements.

This option is useful for detecting aliasing pragma errors. Incorrect aliasing gives runtime errors that are hard to diagnose. When a runtime error occurs, but the error disappears when you use **ignprag** with the **-O** option, the information specified in the aliasing pragmas is likely incorrect.

Syntax
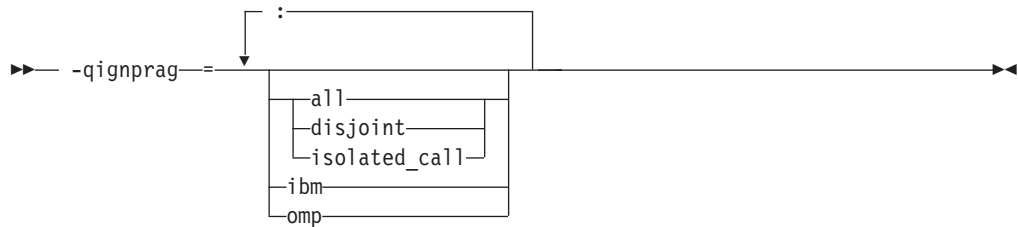
Defaults

Not applicable.

Parameters

all
Ignores all **#pragma isolated_call** and **#pragma disjoint** directives in the source file.

disjoint
Ignores all **#pragma disjoint** directives in the source file.

ibm
▶ C ◀ Ignores all **#pragma ibm snapshot** directives in the source file.

isolated_call
Ignores all **#pragma isolated_call** directives in the source file.

omp
Ignores all OpenMP parallel processing directives in the source file, such as **#pragma omp parallel**, **#pragma omp critical**.

Predefined macros

None.

Examples

To compile myprogram.c and ignore any **#pragma isolated_call** directives, enter:

```
xlc myprogram.c -qignprag=isolated_call
```

Related information
- "#pragma disjoint" on page 341
- "-qisolated_call" on page 182
- "#pragma ibm snapshot" on page 347
- "Pragma directives for parallel processing" on page 377

-qinclude
Category

Input control

Pragma equivalent

None.

Purpose

Specifies additional header files to be included in a compilation unit, as though the files were named in an `#include` statement in the source file.

The headers are inserted before all code statements and any headers specified by an `#include` preprocessor directive in the source file.

This option is provided for portability among supported platforms.

Syntax

►►── -qinclude──=─*file_path*──►◄

Defaults

Not applicable.

Parameters

file_path
    The absolute or relative path and name of the header file to be included in the compilation units being compiled. If *file_path* is specified with a relative path, the search for it follows the sequence described in "Directory search sequence for include files" on page 12.

Usage

-qinclude is applied only to the files specified in the same compilation as that in which the option is specified. It is not passed to any compilations that occur during the link step, nor to any implicit compilations, such as those invoked by the option **-qtemplateregistry**, nor to the files generated by **-qtempinc**.

When the option is specified multiple times in an invocation, the header files are included in order of appearance on the command line. If the same header file is specified multiple times with this option, the header is treated as if included multiple times by `#include` directives in the source file, in order of appearance on the command line.

▶ **C++** When used with **-qtemplateregistry**, **-qinclude** is recorded in the template registry file, along with the source files affected by it. When these file dependencies initiate recompilation of the template registry, the **-qinclude** option is passed to the dependent files only if it had been specified for them when they were added to the template registry.

If you generate a listing file with **-qsource**, the header files included by **-qinclude** do not appear in the source section of the listing. Use **-qshowinc=usr** or **-qshowinc=all** in conjunction with **-qsource** if you want these header files to appear in the listing.

Any pragma directives that must appear before noncommentary statements in a source file will be affected; you cannot use **-qinclude** to include files if you need to preserve the placement of these pragmas.

Predefined macros

None.

Examples

To include the files foo1.h and foo2.h in the source file foo.c, enter:

```
xlc -qinclude=foo1.h foo.c  -qinclude=foo2.h
```

Related information
- "Directory search sequence for include files" on page 12

-qinfo

Category

Error checking and debugging

Pragma equivalent

#pragma options [no]info, #pragma info

Purpose

Produces or suppresses groups of informational messages.

The messages are written to standard output and, optionally, to the listing file if one is generated.

Syntax

Option syntax

```
►►─ -q ─┬─noinfo─────────────────────────────────────┬─►◄
         └─info─┐                                     │
                │          ┌──:──┐                    │
                └─ = ─▼──┬─all────────┬──┐
                         ├─noall──────┤
                         ├─als────────┤
                         ├─noals──────┤
                         ├─group──────┤
                         ├─nogroup────┤
                         ├─private────┤
                         ├─reduction──┤
                         ├─stp────────┤
                         └─nostp──────┘
```

Pragma syntax

```
                              ,
                            ◄─────────────┐
►►─#─pragma─info─(──┬─ all ──────┬──)──────────────────────────►◄
                    ├─ none ──────┤
                    ├─ als ───────┤
                    ├─ noals ─────┤
                    ├─ group ─────┤
                    ├─ nogroup ───┤
                    ├─ private ───┤
                    ├─ reduction ─┤
                    └─ restore ───┘
```
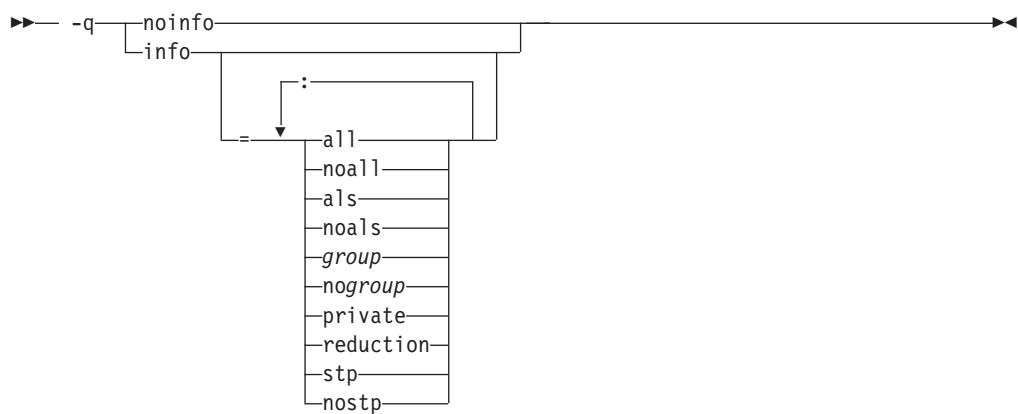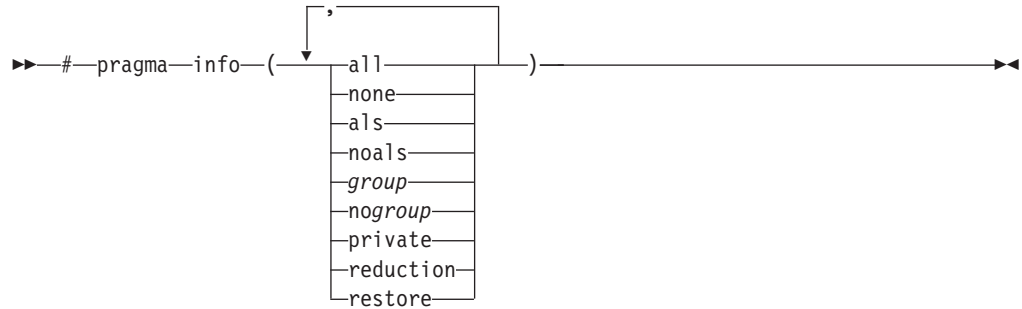
Defaults

-qnoinfo

- **C** -qnoinfo
- **C++** -qinfo=lan:trx

Parameters

all Enables all diagnostic messages for all groups.

noall (option only)
Disables all diagnostic messages for all groups.

none (pragma only)
Disables all diagnostic messages for all groups.

als
Enables reporting possible violations of the ANSI aliasing rule in effect.

noals
Disables reporting possible aliasing-rule violations.

group | **no***group*
Enables or disables specific groups of messages, where *group* can be one or more of:

group
Type of informational messages returned or suppressed.

C **c99** | **noc99**
C code that may behave differently between C89 and C99 language levels.

C++ **cls** | **nocls**
C++ classes.

cmp | **nocmp**
Possible redundancies in `unsigned` comparisons.

cnd | **nocnd**
Possible redundancies or problems in conditional expressions.

cns | **nocns**
Operations involving constants.

cnv | **nocnv**
Conversions.

dcl | **nodcl**
Consistency of declarations.

eff | noeff
Statements and pragmas with no effect.

enu | noenu
Consistency of `enum` variables.

ext | noext
Unused external definitions.

gen | nogen
General diagnostic messages.

gnr | nognr
Generation of temporary variables.

got | nogot
Use of `goto` statements.

ini | noini
Possible problems with initialization.

lan | nolan
Language level effects.

obs | noobs
Obsolete features.

ord | noord
Unspecified order of evaluation.

par | nopar
Unused parameters.

por | nopor
Nonportable language constructs.

ppc | noppc
Possible problems with using the preprocessor.

ppt | noppt
Trace of preprocessor actions.

pro | nopro
Missing function prototypes.

rea | norea
Code that cannot be reached.

ret | noret
Consistency of return statements.

trd | notrd
Possible truncation or loss of data or precision.

tru | notru
Variable names truncated by the compiler.

trx | notrx
Hexadecimal floating point constants rounding.

uni | nouni
Uninitialized variables.

upg | noupg
Generates messages describing new behaviors of the current compiler release as compared to the previous release.

use | nouse
>   Unused auto and static variables.

> ► `C++` **vft | novft**
>   Generation of virtual function tables.

zea | nozea
>   Zero-extent arrays.

private
>   This suboption is deprecated. **-qreport** replaces it. For details, see "-qreport" on
>   page 254 and the "Deprecated options" on page 83 section in the *XL C/C++
>   Compiler Reference*.

reduction
>   This suboption is deprecated. **-qreport** replaces it. For details, see "-qreport" on
>   page 254 and the "Deprecated options" on page 83 section in the *XL C/C++
>   Compiler Reference*.

stp | nostp
>   Issues warnings for procedures that are not protected against stack corruption.
>   **-qinfo=stp** has no effects unless the **-qstackprotect** option is also enabled. Like
>   other **-qinfo** options, **-qinfo=stp** is enabled or disabled through **-qinfo=all /
>   noall**. **-qinfo=nostp** is the default option.

restore (pragma only)
>   Discards the current pragma setting and reverts to the setting specified by the
>   previous pragma directive. If no previous pragma was specified, reverts to the
>   command-line or default option setting.

► `C` Specifying **-qinfo** with no suboptions is equivalent to **-qinfo=all**.

► `C++` Specifying **-qinfo** with no suboptions is equivalent to **-qinfo=all:noppt**.

Usage

Specifying **-qnoinfo** is equivalent to **-qinfo=noall**.

Consider the following when enabling the reporting of aliasing-rule violations:
- **-qalias=ansi** must be set before reporting of aliasing-rule violations (**-qinfo=als**)
  can occur.
- Any level of optimization or inlining implies **-qinfo=noals** and a warning will
  be issued.
- Diagnostics are heuristic and may emit false positives. Points-to analysis cannot
  be evaluated deterministically in static compilation. The points-to analysis used
  for diagnostics is evaluated in a context-and-flow, insensitive manner. The
  sequence of traceback messages in diagnostics is such that if executed in the
  order specified, the indirect expression will point to the offending object. If that
  execution sequence cannot occur in the application, the diagnostic is a false
  positive. (See the **Examples** section for the types of diagnostics that can occur.)

Predefined macros

None.

Examples

To compile `myprogram.c` to produce informational message about all items except conversions and unreached statements, enter:

```
xlc myprogram.c -qinfo=all -qinfo=nocnv:norea
```

▶ C The following example shows code constructs that the compiler detects when the code is compiled with **-qinfo=cnd:eff:got:obs:par:pro:rea:ret:uni** in effect:

```
#define COND 0

void faa() // Obsolete prototype (-qinfo=obs)
{
   printf("In faa\n"); // Unprototyped function call (-qinfo=pro)
}

int foo(int i, int k)
{
   int j; // Uninitialized variable (-qinfo=uni)

   switch(i) {
   case 0:
   i++;
   if (COND) // Condition is always false (-qinfo=cnd)
      i--;   // Unreachable statement (-qinfo=rea)
   break;

   case 1:
      break;
      i++;   // Unreachable statement (-qinfo=rea)
   default:
      k = (i) ? (j) ? j : i : 0;
}

   goto L;   // Use of goto statement (-qinfo=got)
   return 3; // Unreachable statement (-qinfo=rea)
L:
   faa(); // faa() does not have a prototype (-qinfo=pro)

// End of the function may be reached without returning a value
// because of there may be a jump to label L (-qinfo=ret)

} //Parameter k is never referenced (-qinfo=ref)

int main(void) {
({ int i = 0; i = i + 1; i; }); // Statement does not have side effects (-qinfo=eff)

 return foo(1,2);
}
```

▶ C++ The following example shows code constructs that the compiler detects, with this code is compiled with **-qinfo=cls:cnd:eff:use** in effect:

```
#pragma abc       // pragma not supported (-qinfo=eff or -qinfo=gen)

int bar() __attribute__((xyz));    // attribute not supported (-qinfo=eff)
int j();

class A {
   public:
      A():  x(0), y(0), z(0) { };  // this constructor is in the correct order
                                   // hence, no info message.
      A(int m): y(0), z(0)
      { x=m; };              // suggest using member initialization list
                                   for x (-qinfo=cls)
```

```
        A(int m, int n):
        x(0), z(0) { };          // not all data members are initialized
                                 // namely, y is not initialized (-qinfo=cls)

        A(int m, int n, int* l):
        x(m), z(l), y(n) { };        // order of class initialization (-qinfo=cls)

    private:
        int x;
        int y;
        int *z;      // suggest having user-defined copy constructor/
                     // assignment operator to handle the pointer data member
                     // (-qinfo=cls)
};

int foo() {
    int j=5;
    j;        // null statement (-qinfo=eff)
              // The user may mean to call j().

return j;

}

void boo() {
    int x;
    int *i = &x;
    float *f;            // f is not used (-qinfo=use)
    f = (float *) i;     // incompatible type (-qinfo=eff)
                         // With ansi aliasing mode, a float pointer
                         // is not supposed to point to an int
}

void cond(int y) {
    const int i=0;
    int j;
    int k=0;

    if (i) {         // condition is always false (-qinfo=cnd)
            j=3;
     }

    if (1) {         // condition is always true  (-qinfo=cnd)
            j=4;
        }

    j=0;
    if (j==0) {      // cond. is always true  (-qinfo=cnd)
        j=5;
    }

    if (y) {
            k+=5
            }

    if (k==5) {      // This case cannot be determined, because k+=5
                     // is in a conditional block.
              j=6;
              }
}
```

In the following example, the **#pragma info(eff, nouni)** directive preceding
`MyFunction1` instructs the compiler to generate messages identifying statements or
pragmas with no effect, and to suppress messages identifying uninitialized

variables. The **#pragma info(restore)** directive preceding MyFunction2 instructs the compiler to restore the message options that were in effect before the **#pragma info(eff, nouni)** directive was specified.

```
#pragma info(eff, nouni)
int MyFunction1()
{
  .
  .
  .

}

#pragma info(restore)
int MyFunction2()
{
  .
  .
  .

}
```

The following example shows a valid diagnostic for an aliasing violation:

```
t1.c:
int main() {
  short s = 42;
  int *pi = (int*) &s;
  *pi = 63;
  return 0;
}
xlC -+ -qinfo=als t1.c
"t1.c", line 4.3: 1540-0590 (I) Dereference may not conform to the current
                                aliasing rules.
"t1.c", line 4.3: 1540-0591 (I) The dereferenced expression has type "int".
                                "pi" may point to "s" which has incompatible
                                type "short".
"t1.c", line 4.3: 1540-0592 (I) Check assignment at line 3 column 11 of t1.c.
```

In the following example, the analysis is context insensitive in that the two calls to floatToInt are not distinguished. There is no aliasing violation in this example, but a diagnostic is still issued.

```
t2.c:
int* floatToInt(float *pf) { return (int*)pf; }

int main() {
  int i;
  float f;
   int* pi = floatToInt((float*)*&i));
   floatToInt(&f;)
   return *pi;
}

xlC -+ -qinfo=als t2.c
"t2.c", line 8.10: 1540-0590 (I) Dereference may not conform to the current
                                 aliasing rules.
"t2.c", line 8.10: 1540-0591 (I) The dereferenced expression has type "int".
                                 "pi" may point to "f" which has incompatible
                                 type "float".
"t2.c", line 8.10: 1540-0592 (I) Check assignment at line 7 column 14 of t2.c.
"t2.c", line 8.10: 1540-0592 (I) Check assignment at line 1 column 37 of t2.c.
"t2.c", line 8.10: 1540-0592 (I) Check assignment at line 6 column 11 of t2.c.

t3.c:
int main() {
  float f;
  int i = 42;
```

```
   int *p = (int*) &f;
   p = &i;
   return *p;
}

xlC -+ -qinfo=als t3.c
"t3.c", line 6.10: 1540-0590 (I) Dereference may not conform to the current aliasing rules.
"t3.c", line 6.10: 1540-0591 (I) The dereferenced expression has type "int". "p" may point to
        "f" which has incompatible type "float".
"t3.c", line 6.10: 1540-0592 (I) Check assignment at line 4 column 10 of t3.c.
```

Related information
- "-qflag" on page 136
- "-qreport" on page 254
- "-qstackprotect" on page 281
- For a list of deprecated options, see the "Deprecated options" on page 83 section in the *XL C/C++ Compiler Reference*.

-qinitauto

Category

Error checking and debugging

Pragma equivalent

#pragma options [no]initauto

Purpose

Initializes uninitialized automatic variables to a specific value, for debugging purposes.

Syntax

```
                  ┌─noinitauto───────────┐
►►─── -q───┴─initauto──=──hex_value─┴──────────────────────►◄
```

Defaults

-qnoinitauto

Parameters

hex_value
    A two-digit hexadecimal byte value.

Usage

This option generates extra code to initialize the value of automatic variables. It reduces the runtime performance of the program and should only be used for debugging.

Predefined macros

- ▶ C++ __INITAUTO__ is defined to the hex value specified on the **-qinitauto** option or pragma; otherwise, it is undefined.

- ► **C++** __INITAUTO_W__ is defined to the hex value, repeated 4 times, specified on the **-qinitauto** option or pragma; otherwise, it is undefined.

Examples

To compile `myprogram.c` so that automatic variables are initialized to hex value FF (decimal 255), enter:

```
xlc myprogram.c -qinitauto=FF
```

-qinlglue

Category

Object code control

Pragma equivalent

#pragma options [no]inlglue

Purpose

When used with **-O2** or higher optimization, inlines glue code that optimizes external function calls in your application.

Glue code, generated by the linker, is used for passing control between two external functions. When **-qinlglue** is in effect, the optimizer inlines glue code for better performance. When **-qnoinlglue** is in effect, inlining of glue code is prevented.

Note:

This option is ignored as glue code is always generated.

Syntax

```
►►─ -q──┬─noinlglue─┬──────────────────────────────────────────►◄
        └─inlglue───┘
```

Defaults

- -qnoinlglue
- **-qinlglue** when **-qtune=pwr4**, **-qtune=pwr5**, **-qtune=pwr6**, **-qtune=ppc970**, **-qtune=auto**, or **-qtune=balanced** is in effect.

Usage

Inlining glue code can cause the code size to grow. Specifying **-qcompact** overrides the **-qinlglue** setting to prevent code growth. If you want **-qinlglue** to be enabled, do not specify **-qcompact**.

Specifying **-qnoinlglue** or **-qcompact** can degrade performance; use these options with discretion.

The **-qinlglue** option only affects function calls through pointers or calls to an external compilation unit. For calls to an external function, you should specify that the function is imported by using, for example, the **-qprocimported** option.

Predefined macros

None.

Related information
- "-qcompact" on page 114
- "-qprocimported, -qproclocal, -qprocunknown" on page 250
- "-qtune" on page 308

-qinline
Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Attempts to inline functions instead of generating calls to those functions, for improved performance.

Note:
- **-qinline** replaces **-Q** and its suboptions.
- **-Q**, **-Q!**, **-Q**=*threshold*, **-Q+**name*, and **-Q-**name* are all deprecated options and suboptions.
- **-qipa=inline|noinline** and **-qipa=inline** suboptions **auto**, **noauto**, **limit**, and **threshold** are deprecated. **-qinline** replaces them all.
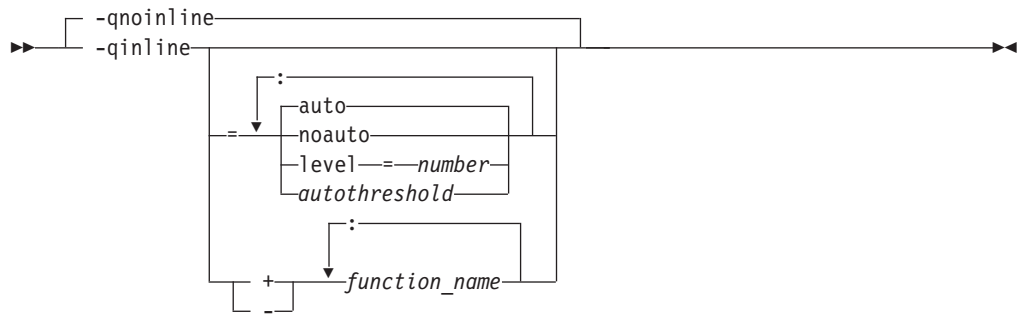
➤ C++ Specifying **-qinline** enables automatic inlining by the compiler front end. Specifying **-qinline** with **-O** provides additional inlining by enabling inlining by the low-level optimizer. In both cases, the compiler attempts to inline all functions, in addition to those defined inside a class declaration or explicitly marked with the `inline` specifier.

➤ C You must specify a minimum optimization level of **-O2** along with **-qinline** to enable inlining of functions, including those declared with the `inline` specifier. You can also use the **-qinline** option to specify restrictions on the functions that should or should not be inlined.

In all cases where **-qinline** is in effect, the compiler uses heuristics to determine whether inlining a specific function will result in a performance benefit. That is, whether a function is appropriate for inlining is subject to limits on the number of inlined calls and the amount of code size increase as a result. Therefore, simply enabling inlining does not guarantee that a given function will be inlined.

Specifying **-qnoinline** disables all inlining, including that performed by the high-level optimizer with the **-qipa** option, and functions declared explicitly as `inline`.

Syntax

```
        ┌─-qnoinline─────────────────────────────────────────┐
►►─┬─────┤                                                    ├─────────────────►◄
   └─-qinline─┬──────────────────────────────────────────────┘
             │          ┌─:◄─────────────┐          │
             │    ┌─▼─┬─auto────────────┬─┐        │
             ├─=──┤   ├─noauto──────────┤ │        │
             │    │   ├─level──=──number─┤ │        │
             │    │   └─autothreshold───┘ │        │
             │    │                       │        │
             │          ┌─:◄────────────┐          │
             │    ┌─+─┐ ┌─▼──────────────┐          │
             └────┤   ├─┴─function_name──┴──────────┘
                  └─-─┘
```

Defaults
- **-qnoinline**
- At an optimization level of **-O0**, the default is **-qinline=noauto**
- At optimization levels of **-O2** and higher, the default is **-qinline=auto**
- **-qinline=auto:level=5** is the default suboption of **-qinline**

Parameters

noauto | auto
> Enables or disables automatic inlining. If you do not specify any **-qinline** suboptions, **-qinline=auto** is the default.

> **Note:** At optimization levels of **-O2** and higher, the default is **-qinline=auto**

level=_number_
> Provides guidance to the compiler about the relative value of inlining. The values you specify for _number_ must be positive integers between 0 and 10 inclusive. The default value for _number_ is 5. If you specify a value less than 5, it implies less inlining. A value greater than 5 implies more inlining than the default.

▶ C   _autothreshold_
> Represents the number of executable statements in a function. The number of executable statements in a function must be fewer than or equal to _autothreshold_ for it to be considered for inlining. The value you specify for _autothreshold_ must be a positive integer. The default value for _autothreshold_ is 20. If you specify a value of 0, no functions are inlined. As you can see in the following example:

```
increment()
{
  int a, b, i;
   for (i=0; i<10; i++) /* statement 1 */
   {
      a=i;              /* statement 2 */
      b=i;              /* statement 3 */
   }
}
```

_function_name_
> Indicates whether the named function should (after +) or should not (after -) be inlined. For example, `-qinline+foo:bar` indicates that procedures foo and bar must be inlined, and `-qinline-bar` indicates that the procedure bar must not be inlined. You cannot mix the "+" and "-" suboptions with each other or with other **-qinline** suboptions. For example, `-qinline+foo-bar` and `-qinline=level=5+foo` are invalid suboption combinations. However, you can use **-qinline** separately to achieve the desired effect. For example, `-qinline+foo:baz -qinline-bar -qinline=noauto:level=7`.

Usage

To maximize inlining, specify optimization (**-O**) and also specify the appropriate **-qinline** options.

Because inlining does not always improve runtime performance, you should test the effects of this option on your code. Do not attempt to inline recursive or mutually recursive functions.

If you specify the **-g** option to generate debug information, inlining may be suppressed.

Predefined macros

None.

Examples

To compile `myprogram.c` so that no functions are inlined, enter:

```
xlc myprogram.c -O2 -qnoinline
```

Assuming you have functions `salary`, `taxes`, `expenses`, and `benefits`, to compile `myprogram.c` so that the compiler tries to inline these functions, you enter:

```
xlc myprogram.c -O2 -qinline+salary:taxes:expenses:benefits
```

If you do not want the functions `salary`, `taxes`, `expenses`, and `benefits` to be inlined when you compile `myprogram.cmyprogram.f`, you enter:

```
xlc myprogram.c -O2 -qinline-salary:taxes:expenses:benefits
```

▶ C In general, you can turn off automatic inlining and request that specific functions be inlined by naming them with the **+** form:

```
-O2 -qinline=noauto -qinline+salary:taxes:benefits
```

This causes only the functions named `salary`, `taxes`, or `benefits` to be inlined, if possible, and no others.

If you want to use the automatic inlining function, you use the `auto` suboption:

```
-O2 -qinline=auto
```

You can specify an inlining level between 6 and 10 to perform more aggressive automatic inlining. For example:

```
-O2 -qinline=auto:level=7
```

If automatic inlining is already enabled by default and you want to specify an inlining level (For example: 7), you enter:

```
-O2 -qinline=level=7
```

Related information
- "-g" on page 149
- "-qipa" on page 176
- "-O, -qoptimize" on page 227
- "The inline function specifier" in the *XL C/C++ Language Reference*
- For a list of deprecated compiler options, see Deprecated options

-qipa

Category

Optimization and tuning
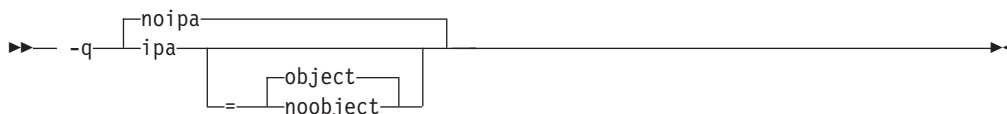
Pragma equivalent

None.

Purpose

Enables or customizes a class of optimizations known as interprocedural analysis (IPA).

IPA is a two-step process: the first step, which takes place during compilation, consists of performing an initial analysis and storing interprocedural analysis information in the object file. The second step, which takes place during linking, and causes a complete recompilation of the entire application, applies the optimizations to the entire program.
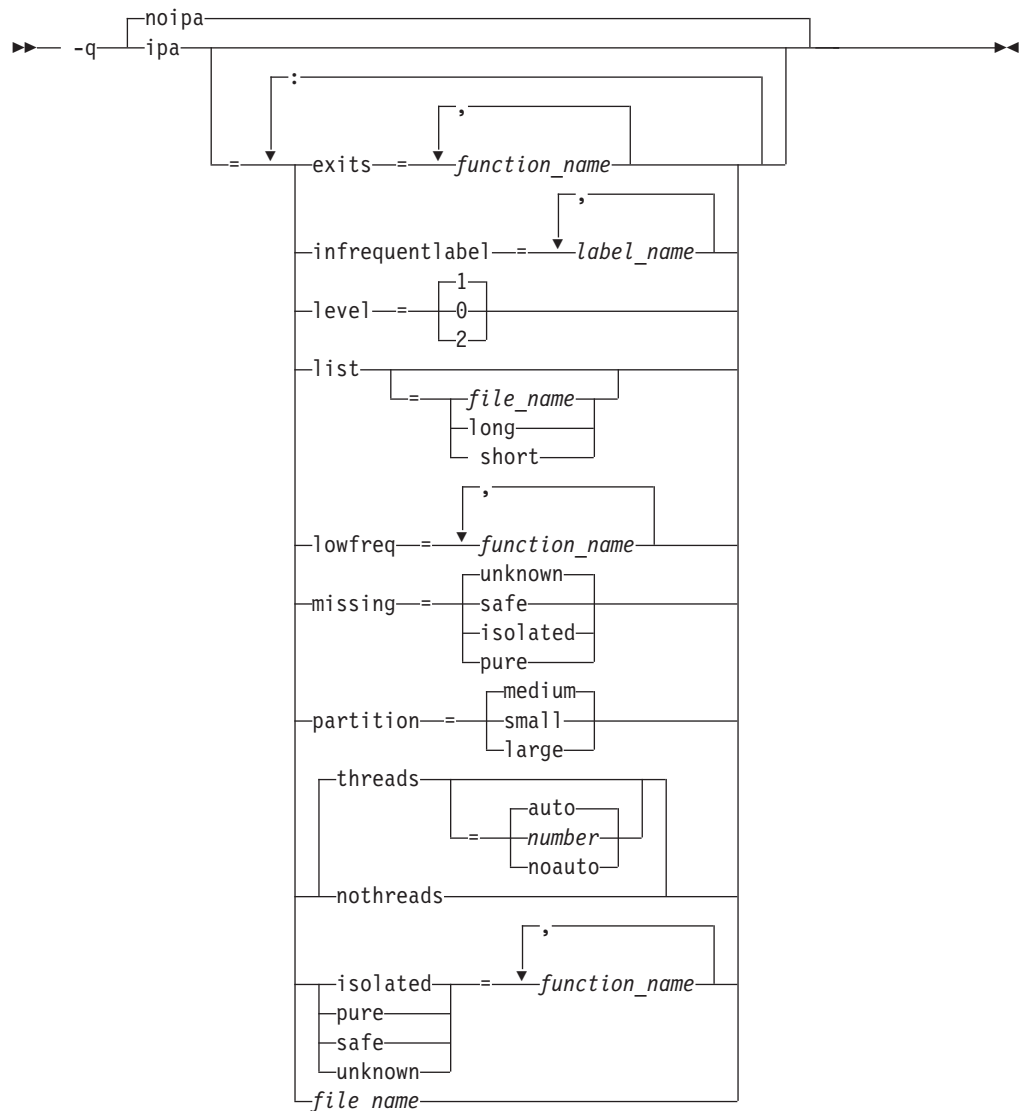
You can use **-qipa** during the compilation step, the link step, or both. If you compile and link in a single compiler invocation, only the link-time suboptions are relevant. If you compile and link in separate compiler invocations, only the compile-time suboptions are relevant during the compile step, and only the link-time suboptions are relevant during the link step.

Syntax

-qipa compile-time syntax

```
                   ┌─noipa─────────────────────┐
►►── -q───────────┼─ipa───────────────────────┼──────────────────►◄
                          ┌─object─┐
                  └─=──────┼────────┼──┘
                          └─noobject─┘
```

-qipa link-time syntax

```
                      ┌─noipa─────────────────────────────────────────────┐
►►── -q ──┬─────────┬─────────────────────────────────────────────────────►◄
          └─ ipa ───┤                                    ┌─:─────────────┐
                    │            ┌──────────────┐         │   ┌─,───────┐ │
                    └─ = ─▼─┬─ exits ─── = ─────▼─ function_name ────────┘
                            │                            ┌─,───────────┐
                            ├─ infrequentlabel ─── = ───▼─ label_name ─┘
                            │                ┌─1─┐
                            ├─ level ─── = ──┼─0─┤
                            │                └─2─┘
                            ├─ list ─────────────────────────┐
                            │        └─ = ──┬─ file_name ─┐
                            │               ├─ long ──────┤
                            │               └─ short ─────┘
                            │                      ┌─,───────────────┐
                            ├─ lowfreq ─── = ─────▼─ function_name ──┘
                            │                ┌─unknown──┐
                            ├─ missing ─── = ─┼─safe─────┤
                            │                ├─isolated─┤
                            │                └─pure─────┘
                            │                  ┌─medium─┐
                            ├─ partition ─── = ─┼─small──┤
                            │                  └─large──┘
                            ├─ threads ──────────────────────┐
                            │          └─ = ──┬─auto───┐
                            │                 ├─number─┤
                            │                 └─noauto─┘
                            ├─ nothreads ─────────────────────
                            │                        ┌─,───────────────┐
                            ├─ isolated ─┬─ = ──────▼─ function_name ──┘
                            ├─ pure ─────┤
                            ├─ safe ─────┤
                            ├─ unknown ──┘
                            └─ file_name ───────────────────
```

Defaults

- **-qnoipa**

Parameters

The following are parameters that may be specified during a separate compile step only:

object | **noobject**

Specifies whether to include standard object code in the output object files.

Specifying **noobject** can substantially reduce overall compile time by not generating object code during the first IPA phase. Note that if you specify **-S** with **noobject**, **noobject** will be ignored.

If compiling and linking are performed in the same step and you do not specify the **-S** or any listing option, **-qipa=noobject** is implied.

Specifying **-qipa** with no suboptions on the compile step is equivalent to **-qipa=object**.

The following are parameters that may be specified during a combined compile and link in the same compiler invocation, or during a separate link step only:

clonearch | noclonearch
This suboption is no longer supported. Consider using **-qtune=balanced**.

cloneproc | nocloneproc
This suboption is no longer supported. Consider using **-qtune=balanced**.

exits
Specifies names of functions which represent program exits. Program exits are calls which can never return and can never call any function which has been compiled with IPA pass 1. The compiler can optimize calls to these functions (for example, by eliminating save/restore sequences), because the calls never return to the program. These functions must not call any other parts of the program that are compiled with **-qipa**.

infrequentlabel
Specifies user-defined labels that are likely to be called infrequently during a program run.

label_name
The name of a label, or a comma-separated list of labels.

isolated
Specifies a comma-separated list of functions that are not compiled with **-qipa**. Functions that you specify as *isolated* or functions within their call chains cannot refer directly to any global variable.

level
Specifies the optimization level for interprocedural analysis. Valid suboptions are one of the following:

0   Performs only minimal interprocedural analysis and optimization.

1   Enables inlining, limited alias analysis, and limited call-site tailoring.

2   Performs full interprocedural data flow and alias analysis.

If you do not specify a level, the default is 1.

To generate data reorganization information, specify the optimization level **-qipa=level=2** or **-O5** together with **-qreport**. During the IPA link phase, the data reorganization messages for program variable data are produced in the data reorganization section of the listing file. Reorganizations include array splitting, array transposing, memory allocation merging, array interleaving, and array coalescing.

list
Specifies that a listing file be generated during the link phase. The listing file contains information about transformations and analyses performed by IPA, as well as an optional object listing for each partition.

If you do not specify a *list_file_name*, the listing file name defaults to a.lst. If you specify **-qipa=list** together with any other option that generates a listing file, IPA generates an a.lst file that overwrites any existing a.lst file. If you have a source file named a.c, the IPA listing will overwrite the regular compiler listing a.lst. You can use the **-qipa=list=**list_file_name suboption to specify an alternative listing file name.

Additional suboptions are one of the following:

short   Requests less information in the listing file. Generates the Object File Map, Source File Map and Global Symbols Map sections of the listing.

long   Requests more information in the listing file. Generates all of the sections generated by the **short** suboption, plus the Object Resolution Warnings, Object Reference Map, Inliner Report and Partition Map sections.

lowfreq
Specifies functions that are likely to be called infrequently. These are typically error handling, trace, or initialization functions. The compiler may be able to make other parts of the program run faster by doing less optimization for calls to these functions.

missing
Specifies the interprocedural behavior of functions that are not compiled with **-qipa** and are not explicitly named in an **unknown**, **safe**, **isolated**, or **pure** suboption.

Valid suboptions are one of the following:

safe   Specifies that the missing functions do not indirectly call a visible (not missing) function either through direct call or through a function pointer.

isolated
Specifies that the missing functions do not directly reference global variables accessible to visible function. Functions bound from shared libraries are assumed to be *isolated*.

pure   Specifies that the missing functions are *safe* and *isolated* and do not indirectly alter storage accessible to visible functions. *pure* functions also have no observable internal state.

unknown
Specifies that the missing functions are not known to be *safe*, *isolated*, or *pure*. This suboption greatly restricts the amount of interprocedural optimization for calls to missing functions.

The default is to assume **unknown**.

partition
Specifies the size of each program partition created by IPA during pass 2. Valid suboptions are one of the following:

- **small**
- **medium**
- **large**

Larger partitions contain more functions, which result in better interprocedural analysis but require more storage to optimize. Reduce the partition size if compilation takes too long because of paging.

pure
Specifies *pure* functions that are not compiled with **-qipa**. Any function specified as *pure* must be *isolated* and *safe*, and must not alter the internal state nor have side-effects, defined as potentially altering any data visible to the caller.

safe

Specifies *safe* functions that are not compiled with **-qipa** and do not call any other part of the program. Safe functions can modify global variables, but may not call functions compiled with **-qipa**.

threads | nothreads

Runs portions of the IPA optimization process during pass 2 in parallel threads, which can speed up the compilation process on multi-processor systems. Valid suboptions for the **threads** suboption are as follows:

auto | noauto

When **auto** is in effect, the compiler selects a number of threads heuristically based on machine load. When **noauto** is in effect, the compiler spawns one thread per machine processor.

number

Instructs the compiler to use a specific number of threads. *number* can be any integer value in the range of 1 to 32 767. However, *number* is effectively limited to the number of processors available on your system.

Specifying **threads** with no suboptions implies **-qipa=threads=auto**.

unknown

Specifies *unknown* functions that are not compiled with **-qipa**. Any function specified as *unknown* can make calls to other parts of the program compiled with **-qipa**, and modify global variables.

file_name

Gives the name of a file which contains suboption information in a special format.

The file format is the following:

```
# ... comment
attribute{, attribute} = name{, name}
missing = attribute{, attribute}
exits = name{, name}
lowfreq = name{, name}
list [ = file-name | short | long ]
level = 0 | 1 | 2
partition = small | medium | large
```

where *attribute* is one of:
- exits
- lowfreq
- unknown
- safe
- isolated
- pure

Note:
- **-qipa=inline** and all of its associated suboptions are deprecated. **-qinline** replaces them all. For details, see "-qinline" on page 173 and "Deprecated options" on page 83.
- As of the V9.0 release of the compiler, the **pdfname** suboption is deprecated; you should use **-qpdf1=***pdfname* or **-qpdf2=***pdfname* in your new applications. See "-qpdf1, -qpdf2" on page 236 for details.

Usage

Specifying **-qipa** automatically sets the optimization level to **-O2**. For additional performance benefits, you can also specify the **-qinline** option. The **-qipa** option extends the area that is examined during optimization and inlining from a single function to multiple functions (possibly in different source files) and the linkage between them.

If any object file used in linking with **-qipa** was created with the **-qipa=noobject** option, any file containing an entry point (the main program for an executable program, or an exported function for a library) must be compiled with **-qipa**.

You can link objects created with different releases of the compiler, but you must ensure that you use a linker that is at least at the same release level as the newer of the compilers used to create the objects being linked.

Some symbols which are clearly referenced or set in the source code may be optimized away by IPA, and may be lost to **debug** or **nm** outputs. Using IPA together with the **-g** compiler will usually result in non-steppable output.

Note that if you specify **-qipa** with **-#**, the compiler does not display linker information subsequent to the IPA link step.

For recommended procedures for using **-qipa**, see "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*.

Predefined macros

None.

Examples

The following example shows how you might compile a set of files with interprocedural analysis:

```
xlc -c *.c -qipaxlc
 -o product *.o -qipa
```

Here is how you might compile the same set of files, improving the optimization of the second compilation, and the speed of the first compile step. Assume that there exist a set of routines, user_trace1, user_trace2, and user_trace3, which are rarely executed, and the routine user_abort that exits the program:

```
xlc -c *.c -qipa=noobject
xlc -c *.o -qipa=lowfreq=user_trace[123]:exit=user_abort
```

Related information
- "-qinline" on page 173
- "-qisolated_call" on page 182
- "-qlibmpi" on page 208
- "#pragma execution_frequency" on page 343
- -qpdf1, -qpdf2
- "-S" on page 262
- Deprecated options
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*
- Runtime environment variables

-qisolated_call
Category

Optimization and tuning

Pragma equivalent

#pragma options isolated_call, #pragma isolated_call

Purpose

Specifies functions in the source file that have no side effects other than those implied by their parameters.

Essentially, any change in the state of the runtime environment is considered a side effect, including:
- Accessing a volatile object
- Modifying an external object
- Modifying a static object
- Modifying a file
- Accessing a file that is modified by another process or thread
- Allocating a dynamic object, unless it is released before returning
- Releasing a dynamic object, unless it was allocated during the same invocation
- Changing system state, such as rounding mode or exception handling
- Calling a function that does any of the above

Marking a function as isolated indicates to the optimizer that external and static variables cannot be changed by the called function and that pessimistic references to storage can be deleted from the calling function where appropriate. Instructions can be reordered with more freedom, resulting in fewer pipeline delays and faster execution in the processor. Multiple calls to the same function with identical parameters can be combined, calls can be deleted if their results are not needed, and the order of calls can be changed.

Syntax
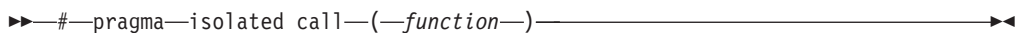
Option syntax

```
►►── -q─isolated_call─=──▼─function─┘──────────────────────────►◄
                         └──:──┐
```

Pragma syntax

```
►►──#──pragma──isolated_call──(──function──)──────────────────►◄
```

Defaults

Not applicable.

Parameters

function

The name of a function that does not have side effects or does not rely on functions or processes that have side effects. *function* is a primary expression that can be an identifier, operator function, conversion function, or qualified name. An identifier must be of type function or a `typedef` of function. ▶ **C++** If the name refers to an overloaded function, all variants of that function are marked as isolated calls.

Usage

The only side effect that is allowed for a function named in the option or pragma is modifying the storage pointed to by any pointer arguments passed to the function, that is, calls by reference. The function is also permitted to examine non-volatile external objects and return a result that depends on the non-volatile state of the runtime environment. Do not specify a function that causes any other side effects; that calls itself; or that relies on local static storage. If a function is incorrectly identified as having no side effects, the program behavior might be unexpected or produce incorrect results.

The **#pragma options isolated_call** directive must be placed at the top of a source file, before any statements. The **#pragma isolated_call** directive can be placed at any point in the source file, before or after calls to the function named in the pragma.

The **-qignprag** compiler option causes aliasing pragmas to be ignored; you can use **-qignprag** to debug applications containing the **#pragma isolated_call** directive.

Predefined macros

None.

Examples

To compile `myprogram.c`, specifying that the functions `myfunction(int)` and `classfunction(double)` do not have side effects, enter:

```
xlc myprogram.c -qisolated_call=myfunction:classfunction
```

The following example shows you when to use the **#pragma isolated_call** directive (on the `addmult` function). It also shows you when not to use it (on the `same` and `check` functions):

```
#include <stdio.h>
#include <math.h>

int addmult(int op1, int op2);
#pragma isolated_call(addmult)

/* This function is a good candidate to be flagged as isolated as its */
/* result is constant with constant input and it has no side effects. */
int addmult(int op1, int op2) {
  int rslt;

  rslt = op1*op2 + op2;
  return rslt;
}

/* The function 'same' should not be flagged as isolated as its state */
/* (the static variable delta) can change when it is called. */
```

```
int same(double op1, double op2) {
  static double delta = 1.0;
  double temp;

  temp = (op1-op2)/op1;
  if (fabs(temp) < delta)
    return 1;
  else {
    delta = delta / 2;
    return 0;
  }
}

/* The function 'check' should not be flagged as isolated as it has a */
/* side effect of possibly emitting output. */
int check(int op1, int op2) {
  if (op1 < op2)
    return -1;
  if (op1 > op2)
    return 1;
  printf("Operands are the same.\n");
  return 0;
}
```

Related information
- "-qignprag" on page 161
- "The const function attribute" and "The pure function attribute" in the *XL C/C++ Language Reference*

-qkeepparm

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

When used with **-O2** or higher optimization, specifies whether function parameters are stored on the stack.

A function usually stores its incoming parameters on the stack at the entry point. However, when you compile code with optimization options enabled, the compiler may remove these parameters from the stack if it sees an optimizing advantage in doing so. When **-qkeepparm** is in effect, parameters are stored on the stack even when optimization is enabled. When **-qnokeepparm** is in effect, parameters are removed from the stack if this provides an optimization advantage.

Syntax

```
                  ┌─nokeepparm─┐
►►── -q───────────┴─keepparm───┴────────────────────────────────►◄
```

Defaults

-qnokeepparm

Usage

Specifying **-qkeepparm** that the values of incoming parameters are available to tools, such as debuggers, by preserving those values on the stack. However, this may negatively affect application performance.

Predefined macros

None.

Related information
- "-O, -qoptimize" on page 227

-qkeyword

Category

Language element control

Pragma equivalent

None

Purpose

Controls whether the specified name is treated as a keyword or as an identifier whenever it appears in your program source.

Syntax

```
►►─ -q──┬─keyword──┬──=─keyword_name─────────────────────────────►◄
        └─nokeyword─┘
```

Defaults

By default all the built-in keywords defined in the C and C++ language standards are reserved as keywords.

Usage

You cannot add keywords to the language with this option. However, you can use **-qnokeyword=**keyword_name to disable built-in keywords, and use **-qkeyword=**keyword_name to reinstate those keywords.

▶ C++ This option can be used with all C++ built-in keywords.

▶ C This option can also be used with the following C keywords:
- asm
- inline
- restrict
- typeof

Note: ▶ C asm is not a keyword when the **-qlanglvl** option is set to **stdc89** or **stdc99**.

Predefined macros

- **C++** __BOOL__ is defined to 1 by default; however, it is undefined when **-qnokeyword=bool** is in effect.
- **C** __C99_INLINE is defined to 1 when **-qkeyword=inline** is in effect.
- __C99_RESTRICT is defined to 1 when **-qkeyword=restrict** is in effect.
- **C** __IBM_GCC_ASM is defined to 1 when **-qkeyword=asm** is in effect. (In C++ it is defined by default.)
- __IBM__TYPEOF__ is defined to 1 when **-qkeyword=typeof** is in effect.

Examples

C++ You can reinstate `bool` with the following invocation:

```
xlc++ -qkeyword=bool
```

C You can reinstate `typeof` with the following invocation:

```
xlc -qkeyword=typeof
```

Related information
- "-qasm" on page 98
- "-qrestrict (C only)" on page 257

-l

Category

Linking

Pragma equivalent

None.

Purpose

Searches for the specified library file, lib*key*.so, and then lib*key*.a for dynamic linking, or just for lib*key*.a for static linking.

Syntax

```
►►── -l─key ──────────────────────────────────────────►◄
```

Defaults

The compiler default is to search only some of the compiler runtime libraries. The default configuration file specifies the default library names to search for with the **-l** compiler option, and the default search path for libraries with the **-L** compiler option.

The C and C++ runtime libraries are automatically added.

Parameters

key
    The name of the library minus the `lib` characters.

Usage

You must also provide additional search path information for libraries not located in the default search path. The search path can be modified with the **-L** option.

The **-l** option is cumulative. Subsequent appearances of the **-l** option on the command line do not replace, but add to, the list of libraries specified by earlier occurrences of **-l**. Libraries are searched in the order in which they appear on the command line, so the order in which you specify libraries can affect symbol resolution in your application.

For more information, refer to the **ld** documentation for your operating system.

Predefined macros

None.

Examples

To compile `myprogram.c` and link it with library `mylibrary` (`libmylibrary.a`) found in the `/usr/mylibdir` directory, enter:

```
xlc myprogram.c -lmylibrary -L/usr/mylibdir
```

Related information
- "-L"
- "Specifying compiler options in a configuration file" on page 7

-L

Category

Linking

Pragma equivalent

None.

Purpose

At link time, searches the directory path for library files specified by the **-l** option.

Syntax

▶▶── -L──*directory_path*──────────────────────────────────◀◀

Defaults

The default is to search only the standard directories. See the compiler configuration file for the directories that are set by default.

Parameters

directory_path
   The path for the directory which should be searched for library files.

Usage

Paths specified with the **-L** compiler option are only searched at link time. To specify paths that should be searched at run time, use the **-R** option.

If the **-L***directory* option is specified both in the configuration file and on the command line, search paths specified in the configuration file are the first to be searched at link time.

For more information, refer to the **ld** documentation for your operating system.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the directory `/usr/tmp/old` is searched for the library `libspfiles.a`, enter:

```
xlc myprogram.c -lspfiles -L/usr/tmp/old
```

Related information
- "-l" on page 186
- "-R" on page 253

-qlanglvl

Category

Language element control

Pragma equivalent

▶ C ◀ #pragma options langlvl, #pragma langlvl

Purpose

Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.

Syntax

-qlanglvl syntax — C

```
                                   :
                              ┌──────────┐
                              ▼        ┌─extc99─┐  ┌─ucs───┐
►►── -q─langlvl──=───────────────────┼─classic─┼──┼─noucs─┼──────────────►◄
                                       ├─extc89──┤
                                       ├─extended┤
                                       ├─saa─────┤
                                       ├─saal2───┤
                                       ├─stdc89──┤
                                       └─stdc99──┘
```
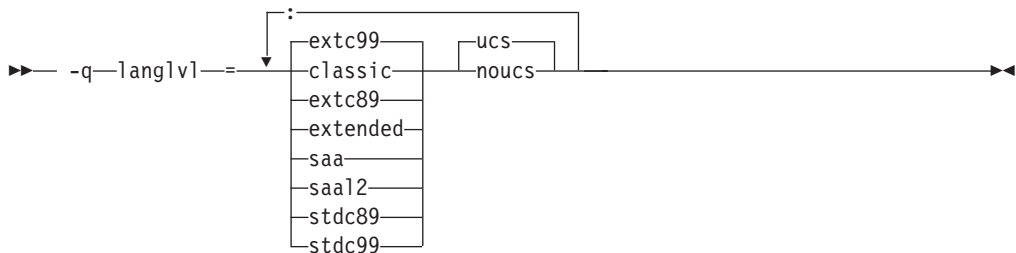
#pragma langlvl syntax — C only

```
►►─#─pragma─langlvl─(─┬─extc99───┬─)─────────────────────────────────►◄
                      ├─classic──┤
                      ├─extc89───┤
                      ├─extended─┤
                      ├─saa──────┤
                      ├─saal2────┤
                      ├─stdc89───┤
                      └─stdc99───┘
```

-qlanglvl syntax — C++

```
                              ┌─:─────────────────────┐
►►── -q─langlvl──=─┬─extended──┬─▼─feature_suboption──┴──────────────►◄
                   └─extended0x─┘
```
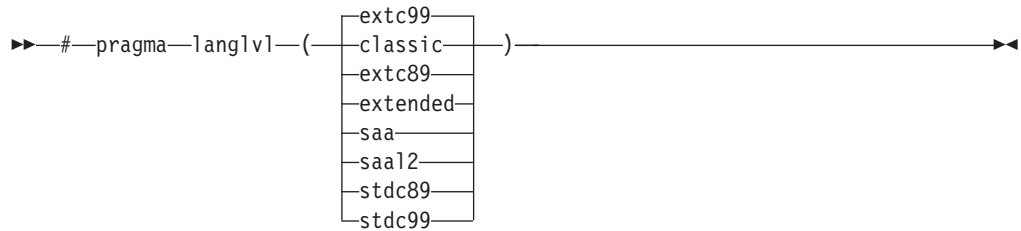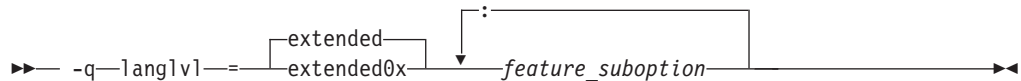
Defaults

- ▶ C The default is set according to the command used to invoke the compiler:
  - **-qlanglvl=extc99:ucs** for the **xlc** and related invocation commands
  - **-qlanglvl=extended:noucs** for the **cc** and related invocation commands
  - **-qlanglvl=stdc89:noucs** for the **c89** and related invocation commands
  - **-qlanglvl=stdc99:ucs** for the **c99** and related invocation commands

- ▶ C++ The suboptions and their default settings for different language levels (compat366, strict98, extended (C++), and extended0x) are listed in Table 24. The default setting On means that the suboption is enabled; otherwise, the default setting Off means that the suboption is disabled.

Table 24. Default Settings of suboptions for different language levels

Options	Language levels			
	compat366	strict98	extended (C++)	extended0x
-qlanglvl=anonstruct | noanonstruct	Off	Off	On	On
-qlanglvl=anonunion | noanonunion	On	Off	On	On
-qlanglvl=ansifor | noansifor	Off	On	On	On
-qlanglvl=ansisinit | noansisinit	On	On	On	On
▶ C++0x -qlanglvl=autotypededuction| noautotypededuction	Off	Off	Off	On
-qlanglvl=c99__func__ | noc99__func__	Off	Off	On	On
-qlanglvl=c99complex |noc99complex	Off	Off	Off	Off
-qlanglvl=c99complexheader | noc99complexheader	Off	Off	Off	Off
-qlanglvl=c99compoundliteral | noc99compoundliteral	Off	Off	On	On
-qlanglvl=c99hexfloat | noc99hexfloat	Off	Off	On	On
▶ C++0x -qlanglvl=c99longlong | noc99longlong	Off	Off	Off	On

*Table 24. Default Settings of suboptions for different language levels (continued)*

Options	Language levels			
	compat366	strict98	extended (C++)	extended0x
▶ C++0x -qlanglvl=c99preprocessor ｜ noc99preprocessor	Off	Off	Off	On
-qlanglvl=c99vla ｜ noc99vla	Off	Off	On	On
-qlanglvl=compatzea ｜ nocompatzea	Off	Off	Off	Off
▶ C++0x -qlanglvl=decltype ｜ nodecltype	Off	Off	Off	On
▶ C++0x -qlanglvl=delegatingctors ｜ nodelegatingctors	Off	Off	Off	On
-qlanglvl=dependentbaselookup ｜ nodependentbaselookup	On	On	On	Off
-qlanglvl=emptystruct ｜ noemptystruct	On	On	On	On
▶ C++0x -qlanglvl=extendedfriend ｜ noextendedfriend	Off	Off	Off	On
▶ C++0x ▶ IBM -qlanglvl=extendedintegersafe ｜ noextendedintegersafe	Off	Off	Off	Off
▶ C++0x -qlanglvl=externtemplate ｜ noexterntemplate	Off	Off	On	On
-qlanglvl=FileScopeConstExternLinkage ｜ noFileScopeConstExternLinkage	Off	Off	Off	Off
▶ C++0x -qlanglvl=inlinenamespace ｜ noinlinenamespace	Off	Off	Off	On
-qlanglvl=gnu_assert ｜ nognu_assert	Off	Off	On	On
-qlanglvl=gnu_complex ｜ nognu_complex	Off	Off	Off	Off
-qlanglvl=gnu_computedgoto ｜ nognu_computedgoto	Off	Off	On	On
-qlanglvl=gnu_explicitregvar ｜ nognu_explicitregvar	Off	Off	On	On
-qlanglvl=gnu_externtemplate ｜ nognu_externtemplate	Off	Off	On	On
-qlanglvl=gnu_labelvalue ｜ nognu_labelvalue	Off	Off	On	On
-qlanglvl=gnu_locallabel ｜ nognu_locallabel	Off	Off	On	On
-qlanglvl=gnu_include_next ｜ nognu_include_next	On	On	On	On
-qlanglvl=gnu_membernamereuse ｜ nognu_membernamereuse	Off	Off	On	On
-qlanglvl=gnu_suffixij ｜ nognu_suffixij	Off	Off	On	On
-qlanglvl=gnu_varargmacros ｜ nognu_varargmacros	Off	Off	On	On
-qlanglvl=gnu_warning ｜ nognu_warning	Off	Off	On	On

Table 24. Default Settings of suboptions for different language levels (continued)

Options	Language levels			
	compat366	strict98	extended (C++)	extended0x
-qlanglvl=illptom ∣ noillptom	On	Off	On	On
-qlanglvl=implicitint ∣ noimplicitint	On	Off	On	On
-qlanglvl=newexcp ∣ nonewexcp	Off	Off	Off	Off
-qlanglvl=offsetnonpod ∣ nooffsetnonpod	On	Off	On	On
-qlanglvl=olddigraph ∣ noolddigraph	Off	On	Off	Off
-qlanglvl=oldfriend ∣ nooldfriend	On	Off	On	Off
-qlanglvl=oldmath ∣ nooldmath	On	Off	Off	Off
-qlanglvl=oldtempacc ∣ nooldtempacc	On	Off	On	On
-qlanglvl=oldtmplalign ∣ nooldtmplalign	On	Off	Off	Off
-qlanglvl=oldtmplspec ∣ nooldtmplspec	On	Off	On	On
-qlanglvl=redefmac ∣ noredefmac	Off	Off	Off	Off
▶ C++0x -qlanglvl=static_assert ∣ nostatic_assert	Off	Off	Off	On
-qlanglvl=trailenum ∣ notrailenum	On	Off	On	On
-qlanglvl=typedefclass ∣ notypedefclass	On	Off	On	On
-qlanglvl=noucs ∣ nonoucs	Off	Off	Off	Off
-qlanglvl=varargmacros ∣ novarargmacros	Off	Off	On	On
▶ C++0x -qlanglvl=variadic[templates] ∣ novariadic[templates]	Off	Off	Off	On
-qlanglvl=zeroextarray ∣ nozeroextarray	Off	Off	On	On

Parameters for C language programs

▶ C The following are the **-qlanglvl**/**#pragma langlvl** parameters for C language programs:

classic
Allows the compilation of nonstandard programs, and conforms closely to the K&R level preprocessor.

The following outlines the differences between the **classic** language level and all other standard-based language levels:

Tokenization
Tokens introduced by macro expansion may be combined with adjacent tokens in some cases. Historically, this was an artifact of the text-based implementations of older preprocessors, and because, in older implementations, the preprocessor was a separate program whose output was passed on to the compiler.

For similar reasons, tokens separated only by a comment may also be combined to form a single token. Here is a summary of how tokenization of a program compiled in **classic** mode is performed:

1. At a given point in the source file, the next token is the longest sequence of characters that can possibly form a token. For example, i+++++j is tokenized as i ++ ++ + j even though i ++ + ++ j may have resulted in a correct program.

2. If the token formed is an identifier and a macro name, the macro is replaced by the text of the tokens specified on its `#define` directive. Each parameter is replaced by the text of the corresponding argument. Comments are removed from both the arguments and the macro text.

3. Scanning is resumed at the first step from the point at which the macro was replaced, as if it were part of the original program.

4. When the entire program has been preprocessed, the result is scanned again by the compiler as in the first step. The second and third steps do not apply here since there will be no macros to replace. Constructs generated by the first three steps that resemble preprocessing directives are not processed as such.

It is in the third and fourth steps that the text of adjacent but previously separate tokens may be combined to form new tokens.

The \ character for line continuation is accepted only in string and character literals and on preprocessing directives.

Constructs such as:

```
#if 0
  "unterminated
#endif
#define US "Unterminating string
char *s = US terminated now"
```

will not generate diagnostic messages, since the first is an unterminated literal in a FALSE block, and the second is completed after macro expansion. However:

```
char *s = US;
```

will generate a diagnostic message since the string literal in US is not completed before the end of the line.

Empty character literals are allowed. The value of the literal is zero.

Preprocessing directives

The # token must appear in the first column of the line. The token immediately following # is available for macro expansion. The line can be continued with \ only if the name of the directive and, in the following example, the (has been seen:

```
#define f(a,b) a+b
f\
(1,2)     /* accepted */

#define f(a,b) a+b
f(\
1,2)      /* not accepted */
```

The rules concerning \ apply whether or not the directive is valid. For example,

```
#\
define M 1   /* not allowed */

#def\
ine M 1      /* not allowed */
```

```
#define\
M 1          /* allowed */

#dfine\
M 1          /* equivalent to #dfine M 1, even
                 though #dfine is not valid  */
```

Following are the preprocessor directive differences.

#ifdef/#ifndef

> When the first token is not an identifier, no diagnostic message is generated, and the condition is FALSE.

#else When there are extra tokens, no diagnostic message is generated.

#endif

> When there are extra tokens, no diagnostic message is generated.

#include

> The < and > are separate tokens. The header is formed by combining the spelling of the < and > with the tokens between them. Therefore /* and // are recognized as comments (and are always stripped), and the " and ' do begin literals within the < and >. (Remember that in C programs, C++-style comments // are recognized when **-qcpluscmt** is specified.)

#line The spelling of all tokens which are not part of the line number form the new file name. These tokens need not be string literals.

#error

> Not recognized.

#define

> A valid macro parameter list consists of zero or more identifiers each separated by commas. The commas are ignored and the parameter list is constructed as if they were not specified. The parameter names need not be unique. If there is a conflict, the last name specified is recognized.

> For an invalid parameter list, a warning is issued. If a macro name is redefined with a new definition, a warning will be issued and the new definition used.

#undef

> When there are extra tokens, no diagnostic message is generated.

Macro expansion

- When the number of arguments on a macro invocation does not match the number of parameters, a warning is issued.
- If the (token is present after the macro name of a function-like macro, it is treated as too few arguments (as above) and a warning is issued.
- Parameters are replaced in string literals and character literals.
- Examples:

```
#define M()    1
#define N(a)   (a)
#define O(a,b) ((a) + (b))

M(); /* no error */
N(); /* empty argument */
O(); /* empty first argument
         and too few arguments */
```

Text output
No text is generated to replace comments.

extc89
Compilation conforms to the ANSI C89 standard, and accepts implementation-specific language extensions.

extc99
Compilation conforms to the ISO C99 standard, and accepts implementation-specific language extensions.

extended
Provides compatibility with the RT compiler and **classic**. This language level is based on C89.

saa
Compilation conforms to the current SAA C CPI language definition. This is currently SAA C Level 2.

saal2
Compilation conforms to the SAA C Level 2 CPI language definition, with some exceptions.

stdc89
Compilation conforms strictly to the ANSI C89 standard, also known as ISO C90.

stdc99
Compilation conforms strictly to the ISO C99 standard.

ucs | noucs (option only)
Controls whether Unicode characters are allowed in identifiers, string literals and character literals in program source code. This suboption is enabled by default when **stdc99** or **extc99** is in effect. For details on the Unicode character set, see "The Unicode standard" in the *XL C/C++ Language Reference*.

The following **-qlanglvl** suboptions are accepted but ignored by the C compiler. Use **extended | extc99 | extc89** to enable the functions that these suboptions imply. For other language levels, the functions implied by these suboptions are disabled.

[no]gnu_assert
GNU C portability option.

[no]gnu_explicitregvar
GNU C portability option.

[no]gnu_include_next
GNU C portability option.

[no]gnu_locallabel
GNU C portability option.

[no]gnu_warning
GNU C portability option.

Parameters for C++ language programs

▶ C++ The following are the **-qlanglvl** group option parameters for corresponding C++ language levels:

extended
Compilation is based on the ISO C++ standard, with some differences to accommodate extended language features.

▶ `C++0x` **extended0x**
Compilation is based on the C++0x standard, invoking all the C++ and currently-supported C++0x features that are implemented in this release. For more information about these C++0x features, see "Extensions for C++0x compatibility" in the *XL C/C++ Language Reference*.

Note: C++0x is a new version of the C++ programming language standard. This is a draft standard and has not been officially adopted in its entirety. Note that future levels of support for this standard are likely to change. The implementation of the language level is based on IBM's interpretation of the draft C++0x standard, and is subject to change at any time without notice. IBM makes no attempt to maintain compatibility with earlier releases, in source or binary, of the new C++0x **-qlanglvl** suboptions (their names or their semantics) and therefore they should not be relied on as a stable programming interface.

The following are the **-qlanglvl** suboption parameters for individual C++ features.

feature_suboption
feature_suboption in the syntax diagram represents a colon-separated list of the remaining C++ options. They can be any of the following:

Note: When multiple **-qlanglvl** group options and suboptions are specified for one individual C++ feature, the last one takes effect.

anonstruct | noanonstruct
Enables or disables support for anonymous structures and classes. Anonymous structures are typically used in unions, as in the following code fragment:

```
union U {
    struct {
        int i:16;
        int j:16;
    };
    int k;
} u;
// ...
u.j=3;
```

When the default, **-qlanglvl=anonstruct**, is in effect, anonymous structures are supported.

This is an extension to the C++ standard and gives behavior that is designed to be compatible with Microsoft® Visual C++. Specify **-qlanglvl=noanonstruct** for compliance with standard C++.

anonunion | noanonunion
Controls the members that are allowed in anonymous unions. When the default, **-qlanglvl=anonunion**, is in effect, anonymous unions can have members of all types that standard C++ allows in non-anonymous unions. For example, non-data members, such as structures, typedefs, and enumerations are allowed. Member functions, virtual functions, or objects of classes that have non-trivial default constructors, copy constructors, or destructors cannot be members of a union, regardless of the setting of this option.

This is an extension to standard C++ and gives behavior that is designed to be compatible with previous versions of VisualAge® C++ and predecessor products, and Microsoft Visual C++. Specify **-qlanglvl=noanonunion** for compliance with standard C++.

ansifor | **noansifor**
Controls whether scope rules defined in the C++ standard apply to names declared in `for` loop initialization statements. When the default, **-qlanglvl=ansifor**, is in effect, standard C++ rules are used, and the following code causes a name lookup error:

```
{
   //...
   for (int i=1; i<5; i++) {
      cout << i * 2 << endl;
   }
   i = 10;  // error
}
```

The reason for the error is that `i`, or any name declared within a `for` loop initialization statement, is visible only within the `for` statement. To correct the error, either declare `i` outside the loop or set **noansifor**.

When **-qlanglvl=noansifor** is in effect, the old language behavior is used; specify **-qlanglvl=noansifor** for compatibility with earlier versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.

ansisinit | **noansisinit**
Controls whether standard C++ rules apply for handling static destructors for global and static objects. When the default, **-qlanglvl=ansisinit**, is in effect, the standard rules are used.

When **-qlanglvl=noansisinit** is in effect, the old language behavior is used; specify **-qlanglvl=noansisinit** for compatibility with earlier versions of VisualAge C++ and predecessor products.

▶ C++0x **autotypededuction**| **noautotypededuction**
Controls whether the auto type deduction feature is enabled. When you specify the **-qlanglvl=autotypededuction** option, the auto type deduction feature is enabled, with which you no longer need to specify a type while declaring a variable. Instead, the compiler deduces the type of an auto variable from the type of its initializer expression.

The **-qlanglvl=autotypededuction** option is included in the group option **-qlanglvl=extended0x**.

The default option is **-qlanglvl=noautotypededuction**.

c99__func__ | **noc99__func__**
Enables or disables support for the C99 `__func__` identifier. For details of this feature, see "func_predefined identifier" in the *XL C/C++ Language Reference*.

c99complex | **noc99complex**
Enables or disables C99 complex data types and related keywords.

c99compoundliteral | **noc99compoundliteral**
Enables or disables support for C99 compound literals.

c99hexfloat | **noc99hexfloat**
Enables or disables support for C99-style hexadecimal floating constants.

▶ C++0x **c99longlong** | **noc99longlong**
Controls whether the C99 `long long` feature is enabled. When you specify the

-qlanglvl=c99longlong option, the C++ compiler provides the C99 `long long` feature, which improves source compatibility between the C and C++ languages.

The **-qlanglvl=c99longlong** option conflicts with the **-qlonglong** option. If you specify both these two options, the **-qlonglong** option is ignored. For more information about the **-qlonglong** option, see "-qlonglong" on page 216.

The **-qlanglvl=c99longlong** option is included in the group option **-qlanglvl=extended0x**, so you can also use this group option to enable the C99 `long long` feature.

The default option is **-qlanglvl=noc99longlong**.

> `C++0x` **c99preprocessor** | **noc99preprocessor**
Controls whether the C99 preprocessor features adopted in C++0x are enabled. When **-qlanglvl=c99preprocessor** is in effect, the C99 and C++0x compilers provide a more common preprocessor interface, which can ease porting C source files to the C++ compiler and avoid preprocessor compatibility issues.

The default option is **-qlanglvl=noc99preprocessor**.

Note: Specifying **-qlanglvl=c99preprocessor** implicitly sets **-qlanglvl=varargmacros**. Also, specifying **-qlanglvl=noc99preprocessor** implicitly sets **-qlanglvl=novarargmacros**.

c99vla | **noc99vla**
Enables or disables support for C99-type variable length arrays.

> `C++0x` **decltype** | **nodecltype**
Controls whether the decltype feature is enabled. With this feature, you can get a type that is based on the resultant type of a possibly type-dependent expression. To enable this feature, you can specify the **-qlanglvl=decltype** option.

The **-qlanglvl=decltype** option is included in the group option **-qlanglvl=extended0x**, so you can also use this group option to enable the decltype feature.

The default option is **-qlanglvl=nodecltype**.

> `C++0x` **delegatingctors** | **nodelegatingctors**
Controls whether the delegating constructors feature is enabled. With this feature, you can concentrate common initializations and post initializations in one constructor, which can make programs more readable and maintainable. To enable this feature, you can specify the **-qlanglvl=delegatingctors** option.

The **-qlanglvl=delegatingctors** option is included in the group option **-qlanglvl=extended0x**, so you can also use this group option to enable the delegating constructors feature.

The default option is **-qlanglvl=nodelegatingctors**.

DependentBaseLookup | **noDependentBaseLookup**
Controls whether the name lookup rules for a template base class of dependent type defined in the Technical Corrigendum 1 (TC1) of the C++ Standard apply. Specify **-qlanglvl=noDependentBaseLookup** for compliance with TC1. When **-qlanglvl=noDependentBaseLookup** is in effect, unqualified names in a template class will not be resolved in a base class if that base class is dependent on a template parameter. These names must be qualified with the

base class name in order to be found by name lookup. When the default, **-qlanglvl=DependentBaseLookup**, is in effect, the behavior of previous XL C++ compilers remains.

The following example shows code that does not compile with **-qlanglvl=noDependentBaseLookup**:

```
struct base
{
        int baseName;
};

template <class B> struct derived : public B
{
   void func()
     {
     int i = baseName;     // this name will not be found in the base class
     };
};

int main(void)
{
   derived<base> x;
   x.func();
   return 0;
}
```

The following example shows code that compiles with or without **-qlanglvl=nodependentbaselookup**:

```
struct base
{
        int baseName;
};

template <class B> struct derived : public B
{
   void func()
     {
     int i = B::baseName;   // qualified name will be found in the base class
     };
};

int main(void)
{
   derived<base> x;
   x.func();
   return 0;
}
```

empty_struct | noempty_struct
This option instructs the compiler to tolerate empty member declarations in structs. Empty member declaration in structs is not allowed. For example, when **-qlanglvl=noemptystruct** is in effect, the following example will be rejected by the compiler:

```
struct S {
     ;  // this line is ill-formed
};
```

The default is **-qlanglvl=noemptystruct**.

▶ C++0x **extendedfriend | noextendedfriend**
Controls whether the extended friend declarations feature is enabled. When you specify the **-qlanglvl=extendedfriend** option, rules governing friend declarations are relaxed as follows:

- Template parameters, `typedef` names, and basic types can be declared as friends.
- The class-key in the context for friend declarations is no longer necessary in C++0x.

The **-qlanglvl=extendedfriend** option is included in the group option **-qlanglvl=extended0x**.

The default option is **-qlanglvl=noextendedfriend**.

Note: **-qlanglvl=extendedfriend** is incompatible with the **-qlanglvl=oldfriend** option. When **-qlanglvl=extendedfriend** is in effect, the **-qlanglvl=oldfriend** option is ignored and the setting of **-qlanglvl=[no]oldfriend** is **-qlanglvl=nooldfriend**.

▶ C++0x   ▶ IBM   **extendedintegersafe** | **noextendedintegersafe**
With this option, if a decimal integer literal that does not have a suffix containing `u` or `U` cannot be represented by the `long long int` type, you can decide whether to use the `unsigned long long int` type to represent the literal or not.

This option takes effect only when the **-qlanglvl=c99longlong** option is specified, otherwise, the compiler issues a warning message to indicate that the option is ignored. When you specify both the **-qlanglvl=c99longlong** and **-qlanglvl=extendedintegersafe** options, if a decimal integer literal that does not have a suffix containing `u` or `U` cannot be represented by the `long long int` type, the compiler issues an error message stating that the value of the literal is out of range.

The default option is **-qlanglvl=noextendedintegersafe** in all the language levels.

▶ C++0x   **externtemplate** | **noexterntemplate**
Controls whether the explicit instantiation declarations feature is enabled. With this feature, you can suppress the implicit instantiations of a template specialization or its members. To enable this feature, you can specify the **-qlanglvl=externtemplate** option, which is the default option.

The **-qlanglvl=externtemplate** option is included in the group options of **-qlanglvl=extended** and **-qlanglvl=extended0x**, so you can use these two group options to enable this feature.

The following table lists options that interact with the **-qlanglvl=externtemplate** option:

Table 25. Options that interact with **-qlanglvl=externtemplate**

Option	Description
-qtemplateregistry, -qtempinc	Explicit instantiation declarations remain effective. Referenced specializations that are the subjects of explicit instantiation declarations, but not the subjects of explicit instantiation definitions in a translation unit are not instantiated from or because of that translation unit.

The following table lists IBM language extensions that interact with the **-qlanglvl=externtemplate** option:

Table 26. IBM language extensions that interact with **-qlanglvl=externtemplate**

IBM language extension	Description
#pragma instantiate	This pragma is semantically the same as an explicit instantiation definition.

Table 26. IBM language extensions that interact with **-qlanglvl=externtemplate** *(continued)*

IBM language extension	Description
#pragma do_not_instantiate	This pragma provides a subset of the functionality of the explicit instantiation declarations which is introduced in the C++0x standard. It is provided for backwards compatibility purposes only. New applications can use explicit instantiation declarations.
#pragma hashome, #pragma ishome	This pragma causes the generation of the virtual function table (VFT) for a class template specialization irrespective of explicit instantiation declarations of the specialization.

The **-qlanglvl=[no]externtemplate** option replaces the deprecated **-qlanglvl=[no]gnu_externtemplate** option. Use the **-qlanglvl=[no]externtemplate** option in your applications.

FileScopeConstExternLinkage | **noFileScopeConstExternLinkage**
Controls whether the file scope of constant variables have internal or external linkage when the static or extern keyword is not specified.

When **-qlanglvl=FileScopeConstExternLinkage** is in effect, all file scope constant variables are marked as externally visible. Otherwise, all file scope constant variables are marked as static.

The default is **-qlanglvl=noFileScopeConstExternLinkage**.

gnu_assert | **nognu_assert**
Enables or disables support for the following GNU C system identification assertions:
- #assert
- #unassert
- #cpu
- #machine
- #system

gnu_complex | **nognu_complex**
Enables or disables GNU complex data types and related keywords.

gnu_computedgoto | **nognu_computedgoto**
Enables or disables support for computed goto statements.

gnu_externtemplate | **nognu_externtemplate**
Enables or disables extern template instantiations. For details of this feature, see "Explicit instantiation" in the *XL C/C++ Language Reference*.

Note: The option **-qlanglvl=[no]gnu_externtemplate** is deprecated in XL C/C++ V11.1; you can use the option **-qlanglvl=[no]externtemplate** instead.

gnu_include_next | **nognu_include_next**
Enables or disables support for the GNU C #include_next preprocessor directive.

gnu_labelvalue | **nognu_labelvalue**
Enables or disables support for labels as values.

gnu_locallabel | **nognu_locallabel**
Enables or disables support for locally-declared labels.

gnu_membernamereuse | **nognu_membernamereuse**
Enables or disables reusing a template name in a member list as a typedef.

gnu_suffixij | nognu_suffixij
  Enables or disables support for GNU-style complex numbers. When
  -qlanglvl=gnu_suffixij is in effect, a complex number can be ended with suffix
  i/I or j/J.

gnu_varargmacros | nognu_varargmacros
  Enables or disables support for GNU-style macros with variable arguments.

  For details of this feature, see "Variadic macro extensions" in the *XL C/C++
  Language Reference*.

gnu_warning | nognu_warning
  Enables or disables support for the GNU C #warning preprocessor directive.

illptom | noillptom
  Controls the expressions that can be used to form pointers to members. When
  the default, **-qlanglvl=illptom**, is in effect, the XL C++ compiler accepts some
  forms that are in common use but do not conform to the C++ Standard. For
  example, the following code defines a pointer to a function member, p, and
  initializes it to the address of C::foo, in the old style:

```
struct C {
void foo(int);
};

void (C::*p) (int) = C::foo;
```

  This is an extension to standard C++ and gives behavior that is designed to be
  compatible with earlier versions of VisualAge C++ and its predecessor
  products, and Microsoft Visual C++.

  Specify **-qlanglvl=noillptom** for compliance with the C++ standard. The
  example code above must be modified to use the & operator.

```
struct C {
void foo(int);
};

void (C::*p) (int) = &C::foo;
```

implicitint | noimplicitint
  Controls whether the compiler accepts missing or partially specified types as
  implicitly specifying int. When the default, **-qlanglvl=implicitint**, is in effect, a
  function declaration at namespace scope or in a member list will implicitly be
  declared to return int. Also, any declaration specifier sequence that does not
  completely specify a type will implicitly specify an integer type. The effect is as
  if the int specifier were present.

  The following specifiers do not completely specify a type:
  • auto
  • const
  • extern
  • extern "*literal*"
  • inline
  • mutable
  • friend
  • register
  • static
  • typedef
  • virtual
  • volatile
  • platform-specific types

> **C++0x** C++0x has removed the use of auto as a storage class specifier. In C++0x, the keyword auto is used as a type specifier. The compiler deduces the type of an auto variable from the type of its initializer expression. For more information, see "The auto type specifier (C++0x)" in the *XL C/C++ Language Reference*.

For example, the return type of function `MyFunction` is `int` because it was omitted in the following code:

```
MyFunction()
{
    return 0;
}
```

Note that any situation where a type is specified is affected by this suboption. This includes, for example, template and parameter types, exception specifications, types in expressions (eg, `casts, dynamic_cast, new`), and types for conversion functions.

This is an extension to the C++ standard and gives behavior that is designed to be compatible with earlier versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.

Specify **-qlanglvl=noimplicitint** for compliance with standard C++. For example, the function declaration above must be modified to:

```
int MyFunction()
{
    return 0;
}
```

> **C++0x** **inlinenamespace** | **noinlinenamespace**
Controls whether inline namespace definitions are enabled, which are namespace definitions preceded by an initial `inline` keyword. A namespace so defined is an inline namespace. When you specify the **-qlanglvl=inlinenamespace** option, members of the inline namespace can be defined and specialized as if they were also members of the enclosing namespace.

The **-qlanglvl=inlinenamespace** option is included in the group option **-qlanglvl=extended0x**.

The default option is **-qlanglvl=noinlinenamespace**.

offsetnonpod | **nooffsetnonpod**
Controls whether the `offsetof` macro can be applied to classes that are not data-only. C++ programmers often casually call data-only classes "Plain Old Data" (POD) classes. When the default, **-qlanglvl=offsetnonpod**, is in effect, you can apply `offsetof` to a class that contains one of the following:
- user-declared constructors or destructors
- user-declared assignment operators
- private or protected non-static data members
- base classes
- virtual functions
- non-static data members of type pointer to member
- a struct or union that has non-data members
- references

This is an extension to the C++ standard, and gives behavior that is designed to be compatible with VisualAge C++ for OS/2 3.0, VisualAge for C++ for Windows®, V3.5, and Microsoft Visual C++. Specify **-qlanglvl=nooffsetnonpod** for compliance with standard C++.

olddigraph | noolddigraph

Enables or disables support for old-style digraphs. When the default, **-qlanglvl=olddigraph**, is in effect, old-style digraphs are not supported. When **-qlanglvl=olddigraph** is in effect, the following digraphs are supported:

Digraph
    Resulting character

%%   # (pound sign)

%%%%

    ## (double pound sign, used as the preprocessor macro concatenation operator)

Specify **-qlanglvl=noolddigraph** for compatibility with standard C++ and the extended C++ language level supported by previous versions of VisualAge C++ and predecessor products.

This suboption only has effect when **-qdigraphs** is in effect.

oldfriend | nooldfriend

Controls whether friend declarations that name classes without elaborated class names are treated as C++ errors. When the default, **-qlanglvl=oldfriend**, is in effect, you can declare a friend class without elaborating the name of the class with the keyword `class`. For example, the statement below declares the class `IFont` to be a friend class:

```
friend IFont;
```

This is an extension to the C++ standard and gives behavior that is designed to be compatible with earlier versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.

Specify the **-qlanglvl=nooldfriend** for compliance with standard C++. The example declaration above must be modified to the following:

```
friend class IFont;
```

Note: -qlanglvl=oldfriend is incompatible with the **-qlanglvl=extendedfriend** option. When **-qlanglvl=extendedfriend** is in effect, the **-qlanglvl=oldfriend** option is ignored and the setting of **-qlanglvl=[no]oldfriend** is **-qlanglvl=nooldfriend**.

oldtempacc | nooldtempacc

Controls whether access to a copy constructor to create a temporary object is always checked, even if creation of the temporary object is avoided. When the default, **-qlanglvl=oldtempacc**, is in effect, access checking is suppressed.

This is an extension to the C++ standard and gives behavior that is designed to be compatible with VisualAge C++ for OS/2 3.0, VisualAge for C++ for Windows, V3.5, and Microsoft Visual C++. Specify **-qlanglvl=nooldtempacc** for compliance with standard C++. For example, the `throw` statement in the following code causes an error because the copy constructor is a protected member of class C:

```
class C {
public:
   C(char *);
protected:
   C(const C&);
};

C foo() {return C("test");} // return copy of C object
```

```
void f()
{
// catch and throw both make implicit copies of
// the throw object
   throw C("error");   // throw a copy of a C object
   const C& r = foo(); // use the copy of a C object
//                              created by foo()
}
```

The example code above contains three ill formed uses of the copy constructor
`C(const C&)`.

oldtmplalign | nooldtmplalign

Controls whether alignment rules specified for nested templates are ignored.
When the default, **-qlanglvl=nooldtmplalign**, is in effect, these alignment rules
are not ignored. For example, given the following template the size of
`A<char>::B` will be 5 with **-qlanglvl=nooldtmplalign**, and 8 with
-qlanglvl=oldtmplalign :

```
template <class T>
struct A {
#pragma options align=packed
 struct B {
  T m;
  int m2;
 };
#pragma options align=reset
};
```

Specify **-qlanglvl=oldtmplalign** for compatibility with VisualAge for C++ V4.0
and predecessor products.

oldtmplspec | nooldtmplspec

Controls whether template specializations that do not conform to the C++
standard are allowed. When the default, **-qlanglvl=oldtmplspec**, is in effect,
you can explicitly specialize a template class as in the following example,
which specializes the template class `ribbon` for type `char`:

```
template<class T> class ribbon { /*...*/};
class ribbon<char> { /*...*/};
```

This is an extension to standard C++ and gives behavior that is designed to be
compatible with VisualAge C++ for OS/2 3.0, VisualAge for C++ for Windows,
V3.5, and Microsoft Visual C++.

Specify **-qlanglvl=nooldtmplspec** for compliance with standard C++. In the
example above, the template specialization must be modified to:

```
template<class T> class ribbon { /*...*/};
template<> class ribbon<char> { /*...*/};
```

redefmac | noredefmac

Controls whether a macro can be redefined without a prior #undef or
`undefine()` statement.

▶ `C++0x` **static_assert | nostatic_assert**

Controls whether the static assertions feature is enabled. When
-qlanglvl=static_assert is in effect, this feature can be used to produce
compile-time assertions for which a severe error message is issued on failure.

-qlanglvl=static_assert is included in the group option **-qlanglvl=extended0x**.

The default is **-qlanglvl=nostatic_assert**.

trailenum | notrailenum

Controls whether trailing commas are allowed in `enum` declarations. When the default, **-qlanglvl=trailenum**, is in effect, one or more trailing commas are allowed at the end of the enumerator list. For example, the following `enum` declaration uses this extension:

```
enum grain { wheat, barley, rye,, };
```

This is an extension to the C++ standard, and is intended to provide compatibility with Microsoft Visual C++.

Specify **-qlanglvl=notrailenum** for compliance with standard C++.

typedefclass | notypedefclass

Controls whether a typedef name can be specified where a class name is expected. When the default, **-qlanglvl=typedefclass**, is in effect, the standard C++ rule applies, and a typedef name cannot be specified where a class name is expected. Specify **-qlanglvl=typedefclass** to allow the use of typedef names in base specifiers and constructor initializer lists, for compatibility with earlier versions of VisualAge for C++ and predecessor products.

ucs | noucs

Controls whether Unicode characters are allowed in identifiers, string literals and character literals in program source code. For details on the Unicode character set, see "The Unicode standard" in the *XL C/C++ Language Reference*.

varargmacros | novarargmacros

Enables or disables support for C99-style variable argument lists in function-like macros.

Note: Specifying **-qlanglvl=c99preprocessor** implicitly set **-qlanglvl=varargmacros**. Vice versa, specifying **-qlanglvl=noc99preprocessor** implicitly set **-qlanglvl=novarargmacros**.
For details of this feature, see "Function-like macros" in the *XL C/C++ Language Reference*.

> **C++0x** **variadic[templates] | novariadic[templates]**

Controls whether the variadic templates feature is enabled. With this feature, you can define class and function templates that have any number (including zero) of parameters. To enable this feature, you can specify the **-qlanglvl=variadic[templates]** option. The word *templates* included in the brackets is optional. If you specify only the **-qlanglvl=variadic** option, the compiler assumes that the **-qlanglvl=variadictemplates** option is specified.

The **-qlanglvl=variadic[templates]** option is included in the group option **-qlanglvl=extended0x**, so you can also use this group option to enable the variadic templates feature.

The default option is **-qlanglvl=novariadic[templates]**.

zeroextarray | nozeroextarray

Controls whether zero-extent arrays are allowed as the last non-static data member in a class definition. When the default, **-qlanglvl=zeroextarray**, is in effect, arrays with zero elements are allowed. The example declarations below define dimensionless arrays `a` and `b`.

```
struct S1 { char a[0]; };
struct S2 { char b[]; };
```

This is an extension to the C++ standard, and is intended to provide compatibility with Microsoft Visual C++.

Specify **-qlanglvl=nozeroextarray** for compliance with standard C++ or with the ANSI language level supported by previous versions of VisualAge C++ and predecessor products.

Usage

▶ `C++` In general, if you specify a suboption with the **no** form of the option, the compiler will diagnose any uses of the feature in your code with a warning, unless you disable the warning with the **-qsuppress** option. Additionally, you can use the **-qinfo=por** option to generate informational messages along with the following suboptions:

- [no]c99complex
- [no]gnu_complex

▶ `C` Since the pragma directive makes your code non-portable, it is recommended that you use the option rather than the pragma. If you do use the pragma, it must appear before any noncommentary lines in the source code. Also, because the directive can dynamically alter preprocessor behavior, compiling with the preprocessing-only options may produce results different from those produced during regular compilation.

Predefined macros

See "Macros related to language levels" on page 397 for a list of macros that are predefined by **-qlanglvl** suboptions.

Related information
- "-qsuppress" on page 291
- "The IBM XL C language extensions" in the *XL C/C++ Language Reference* and "The IBM XL C++ language extensions" in the *XL C/C++ Language Reference*

-qldbl128
Category

Floating-point and integer control

Pragma equivalent

#pragma options [no]ldbl128

Purpose

Increases the size of `long double` types from 64 bits to 128 bits.

Syntax

```
                  ┌─ldbl128───┐
▶▶─ -q──┴─noldbl128─┴──────────────────────────────────────────────────▶◀
```

Defaults

-qldbl128

Usage

The **#pragma options** directive must appear before the first C or C++ statement in the source file, and the option applies to the entire file.

Predefined macros

- __LONGDOUBLE128 and __LONG_DOUBLE_128__ are defined to 1 when **-qldbl128** is in effect; otherwise, they are undefined.
- __LONGDOUBLE64 is defined to 1 when **-qnoldbl128** is in effect; it is undefined when **-qldbl128** is in effect.

Examples

To compile `myprogram.c` so that long double types are 128 bits, enter:

```
xlc myprogram.c -qldbl128
```

-qlib

Category

Linking

Pragma equivalent

None.

Purpose

Specifies whether standard system libraries and XL C/C++ libraries are to be linked.

When **-qlib** is in effect, the standard system libraries and compiler libraries are automatically linked. When **-qnolib** is in effect, the standard system libraries and compiler libraries are not used at link time; only the libraries specified on the command line with the **-l** flag will be linked.

This option can be used in system programming to disable the automatic linking of unneeded libraries.

Syntax

```
         ┌─lib──┐
►►── -q──┴─nolib─┴─────────────────────────────────────────────────►◄
```

Defaults

-qlib

Usage

Using **-qnolib** specifies that no libraries, including the system libraries as well as the XL C/C++ libraries (these are found in the lib/ and lib64/ subdirectories of the compiler installation directory), are to be linked. The system startup files are still linked, unless **-qnocrt** is also specified.

Note that if your program references any symbols that are defined in the standard libraries or compiler-specific libraries, link errors will occur. To avoid these unresolved references when compiling with **-qnolib**, be sure to explicitly link the required libraries by using the command flag **-l** and the library name.

Predefined macros

None.

Examples

To compile `myprogram.c` without linking to any libraries except the compiler library libxlopt.a, enter:

```
xlc myprogram.c -qnolib -lxlopt
```

Related information
- "-qcrt" on page 117

-qlibansi

Category

Optimization and tuning

Pragma equivalent

#pragma options [no]libansi

Purpose

Assumes that all functions with the name of an ANSI C library function are in fact the system functions.

When **libansi** is in effect, the optimizer can generate better code because it will know about the behavior of a given function, such as whether or not it has any side effects.

Syntax

```
►►─ -q─┬─nolibansi─┬──────────────────────────────────────────►◄
        └─libansi───┘
```

Defaults

-qnolibansi

Predefined macros

▶ C++ __LIBANSI__ is defined to 1 when **libansi** is in effect; otherwise, it is not defined.

-qlibmpi

Category

"Optimization and tuning" on page 78

Pragma equivalent

None

Purpose

Asserts that all functions with Message Passing Interface (MPI) names are in fact MPI functions and not a user function with different semantics.

Syntax

```
►►── -q──┬─nolibmpi─┬────────────────────────────────────────►◄
         └─libmpi───┘
```

Defaults

-qnolibmpi

Usage

MPI is a library interface specification for message passing. It addresses the message-passing parallel programming model in which data is moved from the address space of one process to another through cooperative operations. For details about MPI, see the Message Passing Interface Forum.

-qlibmpi allows the compiler to generate better code because it knows about the behavior of a given function, such as whether or not it has any side effects.

When you use **-qlibmpi**, the compiler assumes that all functions with the name of an MPI library function are in fact MPI functions. **-qnolibmpi** makes no such assumptions.

Note: You cannot use this option if your application contains your own version of the library function that is incompatible with the standard one.

Predefined macros

None.

Examples

To compile myprogram.c, enter the following command:

```
xlc -O5 myprogram.c -qlibmpi
```

Related information
- Message Passing Interface Forum
- "-qipa" on page 176

-qlinedebug
Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Generates only line number and source file name information for a debugger.

When **-qlinedebug** is in effect, the compiler produces minimal debugging information, so the resulting object size is smaller than that produced by the **-g** debugging option. You can use the debugger to step through the source code, but you will not be able to see or query variable information. The traceback table, if generated, will include line numbers.

Syntax

```
►►── -q───┬─nolinedebug─┬─────────────────────────────────────────────────────►◄
          └─linedebug───┘
```

Defaults

-qnolinedebug

Usage

When **-qlinedebug** is in effect, function inlining is disabled.

Avoid using **-qlinedebug** with **-O** (optimization) option. The information produced may be incomplete or misleading.

The **-g** option overrides the **-qlinedebug** option. If you specify **-g** with **-qnolinedebug** on the command line, **-qnolinedebug** is ignored and a warning is issued.

Predefined macros

None.

Examples

To compile `myprogram.c` to produce an executable program `testing` so you can step through it with a debugger, enter:

```
xlc myprogram.c -o testing -qlinedebug
```

Related information
- "-g" on page 149
- "-O, -qoptimize" on page 227

-qlist
Category

Listings, messages, and compiler information
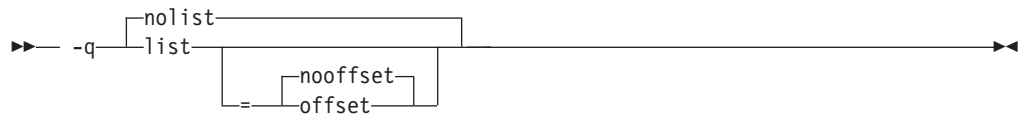
Pragma equivalent

#pragma options [no]list

Purpose

Produces a compiler listing file that includes an object listing.

When **list** is in effect, a listing file is generated with a .lst suffix for each source file named on the command line. For details of the contents of the listing file, see "Compiler listings" on page 19.

You can use the object or assembly listing to help understand the performance characteristics of the generated code and to diagnose execution problems.

Syntax

```
>>-- -q----nolist-----------------------------------------------------><
          '-list-'
                 '-=----nooffset--'
                       '-offset---'
```

Defaults

-qnolist

Parameters

offset | nooffset

Changes the offset of the PDEF header from 00000 to the offset of the start of the text area. Specifying the option allows any program reading the .lst file to add the value of the PDEF and the line in question, and come up with the same value whether **offset** or **nooffset** is specified. The **offset** suboption is only relevant if there are multiple procedures in a compilation unit.

Specifying **list** without the suboption is equivalent to **list=nooffset**.

Usage

The **-qnoprint** compiler option overrides this option.

Predefined macros

None.

Examples

To compile myprogram.c and to produce a listing (.lst) file that includes an object listing, enter:

```
xlc myprogram.c -qlist
```

Related information
- "-qlistopt" on page 214
- "-qprint" on page 248
- "-qsource" on page 276

-qlistfmt

Category

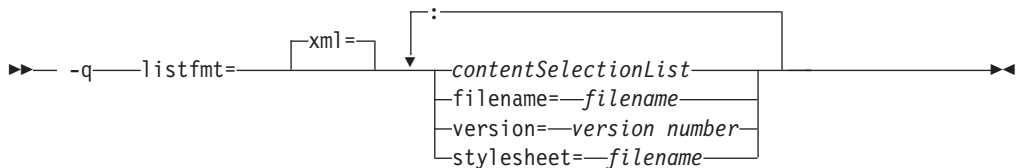Listings, messages, and compiler information

@PROCESS

None.

Pragma equivalent

None.

Purpose

Creates an XML report to assist with finding optimization opportunities.

Syntax

```
►►── -q──listfmt=────┬──xml=──┬──┬─────┬──────────────────────────────────►◄
                     └────────┘  │  :  │
                                 ▼  ┌──────────────────────────┐
                                    ├──contentSelectionList─────┤
                                    ├─filename=─filename─────────┤
                                    ├─version=─version number────┤
                                    └─stylesheet=─filename───────┘
```

Defaults

This option is not on by default. If no *contentSelectionList* options are selected in their positive form, no report is produced. The default is **-qlistfmt=xml=none**.

Parameters

The following list describes **-qlistfmt** parameters:

xml
Indicates that the report should be generated in XML format.

contentSelectionList
The following suboptions provide a filter to limit the type and quantity of information in the report:

data | nodata
Produces data reorganization information.

inlines | noinlines
Produces inlining information.

pdf | nopdf
Produce profile-directed feedback information.

transforms | notransforms
Produces loop transformation information.

all Produces all available report information.

none
Does not produce a report.

filename

Specifies the name of the report file. One file is produced during the compile phase, and one file is produced during the IPA link phase. If no filename is specified, a file with the suffix `.xml` is generated in a way that is consistent with the rules of name generation for the given platform. For example, if compiling `foo.c` the generated XML files are `foo.xml` from the compile step and `a.xml` from the link step.

Note: If you compile and link in one step and use this suboption to specify a file name for the report, the information from the IPA link step will overwrite the information generated during the compile step.

The same will be true if you compile multiple files using the `filename` suboption. The compiler creates an report for each file so the report of the last file compiled will overwrite the previous reports. For example,

`xlc -qlistfmt=xml=all:filename=abc.xml -O3 myfile1.c myfile2.c myfile3.c`

will result in only one report, `abc.xml` based on the compilation of the last file `myfile3.c`

stylesheet

Specifies the name of an existing XML stylesheet for which an `xml-stylesheet` directive is embedded in the resulting report. The default behavior is to not include a stylesheet. The stylesheet shipped with XL C/C++ is `xlstyle.xsl`. This stylesheet renders the XML to an easily read format when viewed using a browser that supports XSLT.

To view the XML report created with the **stylesheet** suboption, you must place the actual stylesheet (`xlstyle.xsl`) and the XML message catalogue (`XMLMessages-`*lang*`.xml` where *lang* refers to the locale set on the compilation machine) in the path specified by the **stylesheet** suboption. For example, if `a.xml` is generated with **stylesheet=xlstyle.xsl**, `xlstyle.xsl` and `XMLMessages-`*lang*`.xml` must be placed into the same directory as `a.xml`, before you can properly view `a.xml` with a browser. The message catalogs and stylesheet are installed in the `/opt/ibmcmp/vacpp/11.1/listings/` directory.

version

If you have written a tool that requires a certain version of this report, you should specify the version. IBM XL C/C++ for Linux, V11.1 creates reports at `v1.0` so if you have written a tool to consume these reports, you should specify `version=v1.0`.

Usage

The information produced in the report by the **-qlistfmt** option depends on which optimization options are used to compiler the program.

- When used with an option that enables inlining such as **-qinline**, the report shows which functions were inlined and why others were not inlined.
- When used with an option that enables loop unrolling, the report contains a summary of how program loops are optimized. The report also includes diagnostic information to show why specific loops cannot be vectorized. For **-qlistfmt** to generate information about loop transformations, you must also specify at least one of the following options:
    - **-qsimd=auto**
    - **-qsmp**
    - **-O5**

- **-qipa=level=2**
- When used with an option that enables parallel transformations, the report contains information about parallel transformations. For **-qlistfmt** to generate information about parallel transformations or parallel performance messages, you must also specify at least one of the following options:
  - **-qsmp**
  - **-O5**
  - **-qipa=level=2**
- When used with the option that enables profiling, **-qpdf**, the report contains information about call and block counts and cache misses.
- When used with an option that produces data reorganizations such as **-qipa=level=2**, the report contains information about those reorganizations.

If no *contentSelectionList* options are selected in their positive form, no report is produced.

Predefined macros

None.

Examples

If you want to compile `myprogram.c` to produce an XML 1.0 report that shows how loops are optimized, enter:

```
xlc -qhot -O3 -qlistfmt=xml=transforms myprogram.c
```

If you want to compile `myprogram.c` to have the XML report show which functions were inlined, enter:

```
xlc -qinline -qlistfmt=xml=inlines myprogram.c
```

Related information
- "-qreport" on page 254
- "Using compiler reports to diagnose optimization opportunities" in the *XL C/C++ Optimization and Programming Guide*

-qlistopt
Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Produces a compiler listing file that includes all options in effect at the time of compiler invocation.

When **listopt** is in effect, a listing file is generated with a .lst suffix for each source file named on the command line. The listing shows options in effect as set by the compiler defaults, the configuration file, and command line settings. For details of the contents of the listing file, see "Compiler listings" on page 19.

Syntax

```
          ┌─nolistopt─┐
►►── -q────┴─listopt──┴──────────────────────────────────────────►◄
```

Defaults

-qnolistopt

Usage

Option settings caused by pragma statements in the program source are not shown in the compiler listing.

The **-qnoprint** compiler option overrides this option.

Predefined macros

None.

Examples

To compile myprogram.c to produce a listing (.lst) file that shows all options in effect, enter:

```
xlc myprogram.c -qlistopt
```

Related information

- "-qlist" on page 210
- "-qprint" on page 248
- "-qsource" on page 276

-qlonglit

Category

Floating-point and integer control

Pragma equivalent

None.

Purpose

In 64-bit mode, when determining the implicit types for integer literals, the compiler behaves as if an l or L suffix were added to integral literals with no suffix or with a suffix consisting only of u or U.

Syntax

```
          ┌─nolonglit─┐
►►── -q────┴─longlit──┴──────────────────────────────────────────►◄
```

Defaults

-qnolonglit

Usage

After you specify the **-qlonglit** option, if the `int` or `unsigned int` type is contained in the implicit type list of a integer literal, the `int` or `unsigned int` type is replaced with the `long int` or `unsigned long int` type, respectively. For more information about the integer literals, see "Integer literals".

Predefined macros

None.

Examples

After you specify the **-qlonglit** option, the integer literal 0x80000000 has the `long int` type in 64-bit mode. Otherwise, if this option is not specified, the integer literal has the `unsigned int` type in both 32-bit and 64-bit modes.

-qlonglong
Category

Language element control

Pragma equivalent

#pragma options [no]longlong

Purpose

Allows IBM `long long` integer types in your program.

Syntax

```
►►─ -q ─┬─ longlong ───┬──────────────────────────────────────────────────► ◄
        └─ nolonglong ─┘
```

Defaults

- ► **C** **-qlonglong** for the xlc, xlc++, xlC, cc and c99 invocation commands; **-qnolonglong** for the c89 invocation command.
- ► **C++** **-qlonglong**

Usage

► **C** This option takes effect when the **-qlanglvl=extended | stdc89 | extc89** option is in effect. It is not valid when the **-qlanglvl=stdc99 | extc99** option is in effect, because the `long long` support provided by this option is incompatible with the semantics of the `long long` types mandated by the C99 standard.

► **C++** This option does not take effect when the **-qlanglvl=c99longlong** option is in effect, because the `long long` support provided by this option is incompatible

with the semantics of the `long long` types mandated by the C99 standard as adopted in C++0x.

Predefined macros

_LONG_LONG is defined to 1 when `long long` data types are available; otherwise, it is undefined.

Examples

To compile `myprogram.c` with support for IBM `long long` integers, enter:

```
cc myprogram.c -qlonglong
```

Related information
- "Integral types" in the *IBM XL C/C++ for Linux, V11.1 Language Reference*

-ma (C only)

See "-qalloca, -ma (C only)" on page 93.

-qmakedep, -M

Category

Output control

Pragma equivalent

None.

Purpose

Creates an output file containing targets suitable for inclusion in a description file for the **make** command.

The output file is named with a .d suffix.

Syntax

```
►►──┬──-M──────────────────────────────────►◄
    └──-q──makedep──┬───────────┤
                    └──=──gcc──┘
```

Defaults

Not applicable.

Parameters

gcc (-qmakedep option only)
  The format of the generated **make** rule to matches the GCC format: the description file includes a single target listing all of the main source file's dependencies.

If you specify -**qmakedep** with no suboption, or -**M**, the description file specifies a separate rule for each of the main source file's dependencies.

Usage

For each source file with a .c, .C, .cpp, or .i suffix named on the command line, an output file is generated with the same name as the object file and a .d suffix. Output files are not created for any other types of input files. If you use the **-o** option to rename the object file, the output file uses the name you specified on the **-o** option. See below for examples.

The output files generated by these options are not **make** files; they must be linked before they can be used with the **make** command. For more information on this command, see your operating system documentation.

The output file contains a line for the input file and an entry for each include file. It has the general form:

file_name.o:*include_file_name*
file_name.o:*file_name.suffix*

You can also use the following option with **qmakedep** and **-M**:

-MF=_file_path_
> Sets the name of the output file, where *file_path* is the full or partial path or file name for the output file. See below for examples.

Include files are listed according to the search order rules for the `#include` preprocessor directive, described in "Directory search sequence for include files" on page 12. If the include file is not found, it is not added to the .d file.

Files with no include statements produce output files containing one line that lists only the input file name.

Predefined macros

None.

Examples

To compile `mysource.c` and create an output file named `mysource.d`, enter:
```
xlc -c -qmakedep mysource.c
```

To compile `foo_src.c` and create an output file named `mysource.d`, enter:
```
xlc -c -qmakedep foo_src.c -MF mysource.d
```

To compile `foo_src.c` and create an output file named `mysource.d` in the `deps/` directory, enter:
```
xlc -c -qmakedep foo_src.c -MF deps/mysource.d
```

To compile `foo_src.c` and create an object file named `foo_obj.o` and an output file named `foo_obj.d`, enter:
```
xlc -c -qmakedep foo_src.c -o foo_obj.o
```

To compile `foo_src.c` and create an object file named `foo_obj.o` and an output file named `mysource.d`, enter:
```
xlc -c -qmakedep foo_src.c -o foo_obj.o -MF mysource.d
```

To compile `foo_src1.c` and `foo_src2.c` to create two output files, named `foo_src1.d` and `foo_src2.d`, respectively, in the `c:/tmp/` directory, enter:

```
xlc -c -qmakedep foo_src1.c foo_src2.c -MF c:/tmp/
```

Related information

- "-MF" on page 223
- "-o" on page 226
- "Directory search sequence for include files" on page 12

-qmaxerr

Category
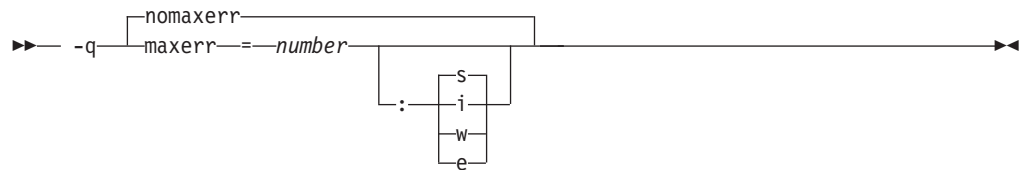
Error checking and debugging
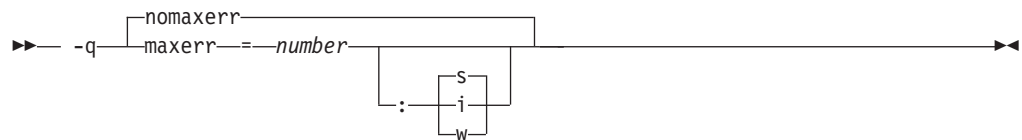
Pragma equivalent

None.

Purpose

Halts compilation when a specified number of errors of a specified severity level or higher is reached.

Syntax

-qmaxerr syntax — C

-qmaxerr syntax — C++

Defaults

-qnomaxerr: The compiler continues to process as much input as possible, until it is not able to generate code.

Parameters

number
 Must be an integer with a value of 1 or greater. An unrecoverable error occurs when the number of errors reaches the limit specified, and compilation stops.

i Specifies a minimum severity level of Informational (I).

w Specifies a minimum severity level of Warning (W).

> **C** **e**

  Specifies a minimum severity level of Error (E).

s Specifies a minimum severity level of Severe error (S).

If you specify **-qmaxerr** with no severity level and the **-qhalt** option or pragma is also in effect, the severity level specified by **halt** is used. If you specify **-qmaxerr** with no severity level and **halt** is not in effect, the default severity level is **s**.

Usage

If the **-qmaxerr** option is specified more than once, the **-qmaxerr** option specified last determines the action of the option. If both the **-qmaxerr** and **-qhalt** options are specified, the **-qmaxerr** or **-qhalt** option specified last determines the severity level used by the **-qmaxerr** option.

Diagnostic messages may be controlled by the **-qflag** option.

Predefined macros

None.

Examples

To stop compilation of `myprogram.c` when 10 warnings are encountered, enter the command:

```
xlc myprogram.c -qmaxerr=10:w
```

To stop compilation of `myprogram.c` when 5 severe errors are encountered, assuming that the current **-qhalt** option value is **s** (severe), enter the command:

```
xlc myprogram.c -qmaxerr=5
```

To stop compilation of `myprogram.c` when 3 informational messages are encountered, enter the command:

```
xlc myprogram.c -qmaxerr=3:i
```

or:

```
xlc myprogram.c -qmaxerr=3 -qhalt=i
```

Related information
- "-qflag" on page 136
- "-qhalt" on page 153
- "Message severity levels and compiler response" on page 17

-qmaxmem
Category

Optimization and tuning

Pragma equivalent

#pragma options maxmem

Purpose

Limits the amount of memory that the compiler allocates while performing specific, memory-intensive optimizations to the specified number of kilobytes.

Syntax

►►── -q─maxmem──=─*size_limit*──►◄

Defaults

- **-qmaxmem=8192** when **-O2** is in effect.
- **-qmaxmem=-1** when **-O3** or higher optimization is in effect.

Parameters

size_limit
  The number of kilobytes worth of memory to be used by optimizations. The limit is the amount of memory for specific optimizations, and not for the compiler as a whole. Tables required during the entire compilation process are not affected by or included in this limit.

  A value of -1 permits each optimization to take as much memory as it needs without checking for limits.

Usage

A smaller limit does not necessarily mean that the resulting program will be slower, only that the compiler may finish before finding all opportunities to increase performance. Increasing the limit does not necessarily mean that the resulting program will be faster, only that the compiler is better able to find opportunities to increase performance if they exist.

Setting a large limit has no negative effect on the compilation of source files when the compiler needs less memory. However, depending on the source file being compiled, the size of subprograms in the source, the machine configuration, and the workload on the system, setting the limit too high, or to -1, might exceed available system resources.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the memory specified for local table is 16384 kilobytes, enter:

```
xlc myprogram.c -qmaxmem=16384
```

-qmbcs, -qdbcs
Category

Language element control

Pragma equivalent

#pragma options [no]mbcs, #pragma options [no]dbcs

Purpose

Enables support for multibyte character sets (MBCS) and Unicode characters in your source code.

When **mbcs** or **dbcs** is in effect, multibyte character literals and comments are recognized by the compiler. When **nombcs** or **nodbcs** is in effect, the compiler treats all literals as single-byte literals.

Syntax

```
                  ┌─nodbcs─┐
                  ├─nombcs─┤
►►── -q───────────┼─mbcs───┤──────────────────────────────────────────◄◄
                  └─dbcs───┘
```

Defaults

-qnombcs, -qnodbcs

Usage

For rules on using multibyte characters in your source code, see "Multibyte characters" in the *XL C/C++ Language Reference*.

In addition, you can use multibyte characters in the following contexts:

- In file names passed as arguments to compiler invocations on the command line; for example:

  xlc /u/myhome/c_programs/kanji_files/*multibyte_char*.c -o*multibyte_char*

- In file names, as suboptions to compiler options that take file names as arguments

- In the definition of a macro name using the **-D** option; for example:

  -DMYMACRO="kps*multibyte_char*dcs"
  -DMYMACRO='*multibyte_char*'

Listing files display the date and time for the appropriate international language, and multibyte characters in the source file name also appear in the name of the corresponding list file. For example, a C source file called:

multibyte_char.c

gives a list file called

multibyte_char.lst

Predefined macros

None.

Examples

To compile `myprogram.c` if it contains multibyte characters, enter:

```
xlc myprogram.c -qmbcs
```

Related information
• "-D" on page 120

-MF

Category

Output control

Pragma equivalent

None.

Purpose

Specifies the target for the output generated by the **-qmakedep** or **-M** options.

This option is used only together with the **-qmakedep** or **-M** options. See the description for the "-qmakedep, -M" on page 217 for more information.

Syntax

▶▶── -MF─*path*──◀◀

Defaults

Not applicable.

Parameters

path
> The target output path. *path* can be a full directory path or file name. If *path* is the name of a directory, the dependency file generated by the compiler is placed into the specified directory. If you do not specify a directory, the dependency file is stored in the current working directory.

Usage

If the file specified by **-MF** option already exists, it will be overwritten.

If you specify a single file name for the **-MF** option when compiling multiple source files, only a single dependency file will be generated containing the **make** rule for the last file specified on the command line.

Predefined macros

None.

Related information
• "-qmakedep, -M" on page 217
• "-o" on page 226

- "Directory search sequence for include files" on page 12

-qminimaltoc

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Controls the generation of the table of contents (TOC), which the compiler creates for an executable file in 64-bit compilation mode.

Programs compiled in 64-bit mode have a limit of 8192 TOC entries. As a result, you may encounter "relocation truncation" error messages when linking large programs in 64-bit mode; these error messages are caused by TOC overflow conditions. When **-qminimaltoc** is in effect, the compiler avoids these overflow conditions by placing TOC entries into a separate data section for each object file.

Specifying **-qminimaltoc** ensures that the compiler creates only one TOC entry for each compilation unit. Specifying this option can minimize the use of available TOC entries, but its use impacts performance. Use the **-qminimaltoc** option with discretion, particularly with files that contain frequently executed code.

Syntax

```
              ┌─nominimaltoc─┐
►►── -q───────┴─minimaltoc───┴──────────────────────────────────►◄
```

Defaults

-qnominimaltoc

Usage

This compiler option applies to 64-bit compilations only.

Compiling with **-qminimaltoc** may create slightly slower and larger code for your program. However, these effects may be minimized by specifying optimizing options when compiling your program.

Predefined macros

None.

-qmkshrobj

Category

Output control

Pragma equivalent

None.

Purpose

Creates a shared object from generated object files.

Use this option, together with the related options described later in this topic, instead of calling the linker directly to create a shared object. The advantages of using this option are the automatic handling of link-time C++ template instantiation (using either the template include directory or the template registry), and compatibility with **-qipa** link-time optimizations (such as those performed at **-O5**).

Syntax

-qmkshrobj syntax

```
►►─── -q─mkshrobj──────────────────────────────────────────────────►◄
```

Defaults

By default, the output object is linked with the runtime libraries and startup routines to create an executable file.

Usage

Specifying **-qmkshrobj** implies **-qpic**.

You can also use the following related options with **-qmkshrobj**:

-o *shared_file*
    The name of the file that holds the shared file information. The default is a.out.

-e *name*
    Sets the entry name for the shared executable to *name*.

Note: Options **-qmkshrobj** and **-qstaticlink** are incompatible and cannot be specified together. If you specify **-qmkshrobj** and **-qstaticlink** (or **-qstaticlink=libgcc**) together, the driver issues the following message: 1501–264 `The options $1 and $2 are incompatible. Option $1 is ignored.`

For detailed information about using **-qmkshrobj** to create shared libraries, see "Constructing a library" in the *XL C/C++ Optimization and Programming Guide*.

Predefined macros

None.

Examples

To construct the shared library `big_lib.so` from three smaller object files, enter the following command:
```
xlc -qmkshrobj -o big_lib.so lib_a.o lib_b.o lib_c.o
```

Related information
- "-e" on page 126
- "-qipa" on page 176
- "-o"
- "-qpic" on page 243
- "-qpriority (C++ only)" on page 249

-o

Category

Output control

Pragma equivalent

None.

Purpose

Specifies a name for the output object, assembler, or executable file.

Syntax

►►── -o─*path*──►◄

Defaults

See "Types of output files" on page 4 for the default file names and suffixes produced by different phases of compilation.

Parameters

path
> When you are using the option to compile from source files, *path* can be the name of a file or directory. The *path* can be a relative or absolute path name. When you are using the option to link from object files, *path* must be a file name.
>
> If the *path* is the name of an existing directory, files created by the compiler are placed into that directory. If *path* is not an existing directory, the *path* is the name of the file produced by the compiler. See below for examples.
>
> You can not specify a file name with a C or C++ source file suffix (.C, .c, .cpp, or .i), such as myprog.c or myprog.i; this results in an error and neither the compiler nor the linker is invoked.

Usage

If you use the **-c** option with **-o** together and the *path* is not an existing directory, you can only compile one source file at a time. In this case, if more than one source file name is listed in the compiler invocation, the compiler issues a warning message and ignores **-o**.

The **-E**, **-P**, and **-qsyntaxonly** options override the **-o** option.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the resulting executable is called `myaccount`, assuming that no directory with name `myaccount` exists, enter:

```
xlc myprogram.c -o myaccount
```

To compile `test.c` to an object file only and name the object file `new.o`, enter:

```
xlc test.c -c -o new.o
```

Related information
- "-c" on page 105
- "-E" on page 127
- "-P" on page 232
- "-qsyntaxonly (C only)" on page 293

-O, -qoptimize
Category

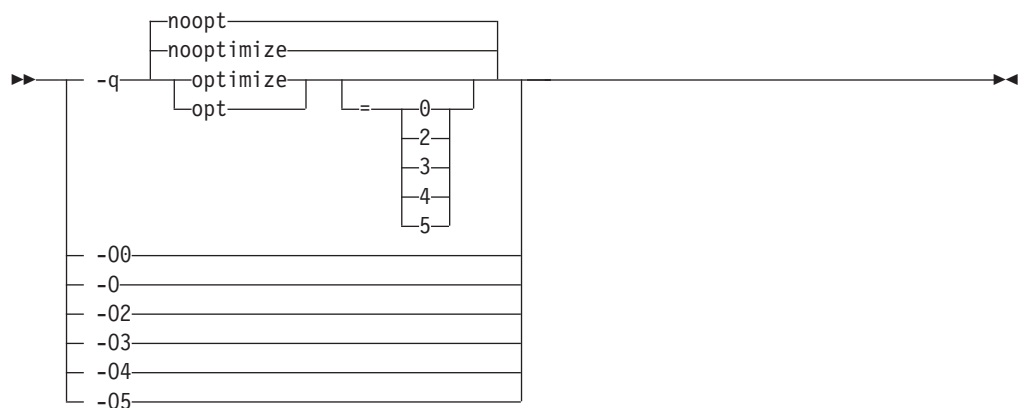Optimization and tuning

Pragma equivalent

#pragma options [no]optimize

Purpose

Specifies whether to optimize code during compilation and, if so, at which level.

Syntax

Defaults

-qnooptimize or **-O0** or **-qoptimize=0**

Parameters

-O0 | nooptimize | noopt | optimize|opt=0

Performs only quick local optimizations such as constant folding and elimination of local common subexpressions.

This setting implies **-qstrict_induction** unless **-qnostrict_induction** is explicitly specified.

-O | -O2 | optimize | opt | optimize|opt=2

Performs optimizations that the compiler developers considered the best combination for compilation speed and runtime performance. The optimizations may change from product release to release. If you need a specific level of optimization, specify the appropriate numeric value.

This setting implies **-qstrict** and **-qnostrict_induction**, unless explicitly negated by **-qstrict_induction** or **-qnostrict**.

-O3 | optimize|opt=3

Performs additional optimizations that are memory intensive, compile-time intensive, or both. They are recommended when the desire for runtime improvement outweighs the concern for minimizing compilation resources.

-O3 applies the **-O2** level of optimization, but with unbounded time and memory limits. **-O3** also performs higher and more aggressive optimizations that have the potential to slightly alter the semantics of your program. The compiler guards against these optimizations at **-O2**. The aggressive optimizations performed when you specify **-O3** are:

1. Aggressive code motion, and scheduling on computations that have the potential to raise an exception, are allowed.

   Loads and floating-point computations fall into this category. This optimization is aggressive because it may place such instructions onto execution paths where they *will* be executed when they *may* not have been according to the actual semantics of the program.

   For example, a loop-invariant floating-point computation that is found on some, but not all, paths through a loop will not be moved at **-O2** because the computation may cause an exception. At **-O3**, the compiler will move it because it is not certain to cause an exception. The same is true for motion of loads. Although a load through a pointer is never moved, loads off the static or stack base register are considered movable at **-O3**. Loads in general are not considered to be absolutely safe at **-O2** because a program can contain a declaration of a static array `a` of 10 elements and load `a[60000000003]`, which could cause a segmentation violation.

   The same concepts apply to scheduling.

   Example:

   In the following example, at **-O2**, the computation of `b+c` is not moved out of the loop for two reasons:

   - It is considered dangerous because it is a floating-point operation
   - `t` does not occur on every path through the loop

   At **-O3**, the code is moved.

   ```
       ...
       int i ;
       float a[100], b, c ;
       for (i = 0 ; i < 100 ; i++)
        {
   ```

```
        if (a[i] < a[i+1])
         a[i] = b + c ;
        }
         ...
```

2. Conformance to IEEE rules are relaxed.

   With **-O2** certain optimizations are not performed because they may produce an incorrect sign in cases with a zero result, and because they remove an arithmetic operation that may cause some type of floating-point exception.

   For example, **X + 0.0** is not folded to X because, under IEEE rules, **-0.0 + 0.0 = 0.0**, which is **-X**. In some other cases, some optimizations may perform optimizations that yield a zero result with the wrong sign. For example, **X - Y * Z** may result in a **-0.0** where the original computation would produce **0.0**.

   In most cases the difference in the results is not important to an application and **-O3** allows these optimizations.

3. Floating-point expressions may be rewritten.

   Computations such as **a*b*c** may be rewritten as **a*c*b** if, for example, an opportunity exists to get a common subexpression by such rearrangement. Replacing a divide with a multiply by the reciprocal is another example of reassociating floating-point computations.

4. Specifying **-O3** implies **-qhot=level=0**, unless you explicitly specify **-qhot** or **-qhot=level=1** option.

   -qfloat=rsqrt is set by default with **-O3**.

   -qmaxmem=-1 is set by default with **-O3**, allowing the compiler to use as much memory as necessary when performing optimizations.

   Built-in functions do not change `errno` at **-O3**.

   Integer divide instructions are considered too dangerous to optimize even at **-O3**.

   Refer to "-qflttrap" on page 141 to see the behavior of the compiler when you specify **optimize** options with the **-qflttrap** option.

   You can use the **-qstrict** and **-qstrict_induction** compiler options to turn off effects of **-O3** that might change the semantics of a program. Specifying **-qstrict** together with **-O3** invokes all the optimizations performed at **-O2** as well as further loop optimizations. Reference to the **-qstrict** compiler option can appear before or after the **-O3** option.

   The **-O3** compiler option followed by the **-O** option leaves **-qignerrno** on.

   When **-O3** and **-qhot=level=1** are in effect, the compiler replaces any calls in the source code to standard math library functions with calls to the equivalent MASS library functions, and if possible, the vector versions.

-O4 | optimize|opt=4

This option is the same as **-O3**, except that it also:

- Sets the **-qarch** and **-qtune** options to the architecture of the compiling machine
- Sets the **-qcache** option most appropriate to the characteristics of the compiling machine
- Sets the **-qhot** option
- Sets the **-qipa** option

Note: Later settings of **-O**, **-qcache**, **-qhot**, **-qipa**, **-qarch**, and **-qtune** options will override the settings implied by the **-O4** option.

-O5 | optimize|opt=5
This option is the same as **-O4**, except that it:

- Sets the **-qipa=level=2** option to perform full interprocedural data flow and alias analysis.

Note:

Later settings of **-O**, **-qcache**, **-qipa**, **-qarch**, and **-qtune** options will override the settings implied by the **-O5** option.

Usage

Increasing the level of optimization may or may not result in additional performance improvements, depending on whether additional analysis detects further opportunities for optimization.

Compilations with optimizations may require more time and machine resources than other compilations.

Optimization can cause statements to be moved or deleted, and generally should not be specified along with the **-g** flag for debugging programs. The debugging information produced may not be accurate.

Predefined macros

- __OPTIMIZE__ is predefined to 2 when **-O | O2** is in effect; it is predefined to 3 when **-O3 | O4 | O5** is in effect. Otherwise, it is undefined.
- __OPTIMIZE_SIZE__ is predefined to 1 when **-O | -O2 | -O3 | -O4 | -O5** and **-qcompact** are in effect. Otherwise, it is undefined.

Examples

To compile and optimize myprogram.c, enter:

```
xlc myprogram.c -O3
```

Related information

- "-qhot" on page 155
- "-qipa" on page 176
- "-qpdf1, -qpdf2" on page 236
- "-qstrict" on page 286
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*.

-qoptdebug
Category

Error checking and debugging

Pragma equivalent

None.

Purpose

When used with high levels of optimization, produces files containing optimized pseudocode that can be read by a debugger.

An output file with a .optdbg extension is created for each source file compiled with **-qoptdebug**. You can use the information contained in this file to help you understand how your code actually behaves under optimization.

Syntax

```
►►──-q──┬─nooptdebug─┬──────────────────────────────────────►◄
        └─optdebug───┘
```

Defaults

-qnooptdebug

Usage

-qoptdebug only has an effect when used with an option that enables the high-level optimizer, namely **-O3** or higher optimization level, or **-qhot**, **-qsmp**, **-qipa**, or **-qpdf**. You can use the option on both compilation and link steps. If you specify it on the compile step, one output file is generated for each source file. If you specify it on the **-qipa** link step, a single output file is generated.

You must still use the **-g** or **-qlinedebug** option to include debugging information that can be used by a debugger.

For more information and examples of using this option, see "Using -qoptdebug to help debug optimized programs" in the *XL C/C++ Optimization and Programming GuideXL C/C++ Optimization and Programming Guide*.

Related information

- "-O, -qoptimize" on page 227
- "-qhot" on page 155
- "-qipa" on page 176
- "-qpdf1, -qpdf2" on page 236
- "-qsmp" on page 272
- "-g" on page 149
- "-qlinedebug" on page 209

-p, -pg, -qprofile
Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Prepares the object files produced by the compiler for profiling.

When you compile with a profiling option, the compiler produces monitoring code that counts the number of times each routine is called. The compiler replaces the startup routine of each subprogram with one that calls the monitor subroutine at the start. When you execute the compiled program and it ends normally, it writes the recorded information to a gmon.out file. You can then use the **gprof** command to generate a runtime profile.

Syntax

```
►►─┬──────── -p ────────────────────────────────────────────────►◄
   ├──────── -pg ────────────┤
   └─ -q ─ profile ─ = ─┬─ p ──┤
                        └─ pg ─┘
```

Defaults

Not applicable.

Usage

When you are compiling and linking in separate steps, you must specify the profiling option in both steps.

If the **-qtbtable** option is not set, the profiling options will generate full traceback tables.

Predefined macros

None.

Examples

To compile `myprogram.c` to include profiling data, enter:

```
xlc myprogram.c -p
```

Remember to compile *and* link with one of the profiling options. For example:

```
xlc myprogram.c -p -c
xlc myprogram.o -p -o program
```

Related information
- "-qtbtable" on page 296
- See your operating system documentation for more information on the **gprof** command.

-P

Category

Output control

Pragma equivalent

None.

Purpose

Preprocesses the source files named in the compiler invocation, without compiling, and creates an output preprocessed file for each input file.

The preprocessed output file has the same name as the input file, with an .i suffix.

Syntax

►►─── -P──►◄

Defaults

By default, source files are preprocessed, compiled, and linked to produce an executable file.

Usage

The **-P** option accepts any file name, except those with an .i suffix. Otherwise, source files with unrecognized file name suffixes are treated and preprocessed as C files, and no error message is generated.

Unless **-qppline** is specified, `#line` directives are not generated.

Line continuation sequences are removed and the source lines are concatenated.

The **-P** option retains all white space including line-feed characters, with the following exceptions:
- All comments are reduced to a single space (unless **-C** is specified).
- Line feeds at the end of preprocessing directives are not retained.
- White space surrounding arguments to function-style macros is not retained.

The **-P** option is overridden by the **-E** option. The **-P** option overrides the **-c**, **-o**, and **-qsyntaxonly** option.

Predefined macros

None.

Related information
- "-C, -C!" on page 106
- "-E" on page 127
- "-qppline" on page 244
- "-qsyntaxonly (C only)" on page 293

-qpack_semantic
Category

Portability and migration

Pragma equivalent

None.

Purpose

Controls the syntax and semantics of the **#pragma pack** directive.

Syntax

```
►►── -q──pack_semantic──=──┬──ibm──┬──────────────────────────────────────►◄
                           └──gnu──┘
```

Defaults

-qpack_semantic=ibm

Parameters

gnu
> Uses the GCC syntax and semantics for **#pragma pack**. The effects of this suboption are:
> * The current packing value is independent from the packing stack.
> * Values are only placed on the pack stack with the **push** parameter. Only values that have been specified with the **push** parameter can be removed from the stack by the **pop** parameter.
> * If a **#pragma pack** directive is specified inside a nested aggregate, it affects the outer, containing aggregate as well.

ibm
> Uses the IBM syntax and semantics for **#pragma pack**. The effects of this suboption are:
> * The current packing value is automatically placed at the top of the packing stack.
> * There is no **push** parameter. Any value can be removed from the stack by the **pop** parameter.
> * If a **#pragma pack** directive is specified inside a nested aggregate, it only affects aggregates that follow it; that is, it can only affected the inner aggregates.

See "#pragma pack" on page 360 for full details on the syntax and semantics, as well as examples, of these suboptions.

Usage

You should not need to use this option unless you are porting applications compiled with GCC and need to preserve source-level compatibility with the GCC version of the pragma directive.

Predefined macros

None.

Examples

See "#pragma pack" on page 360 for examples.

-qpath

Category

Compiler customization

Pragma equivalent

None.

Purpose

Determines substitute path names for XL C/C++ executables such as the compiler, assembler, and linker.

You can use this option if you want to keep multiple levels of some or all of the XL C/C++ executables and have the option of specifying which one you want to use. This option is preferred over the **-B** and **-t** options.

Syntax

```
►►── -q─path──=──┬─a─┬──:──directory_path──────────────────────────►◄
                 ├─b─┤
                 ├─c─┤
                 ├─C─┤
                 ├─d─┤
                 ├─I─┤
                 ├─L─┤
                 └─l─┘
```
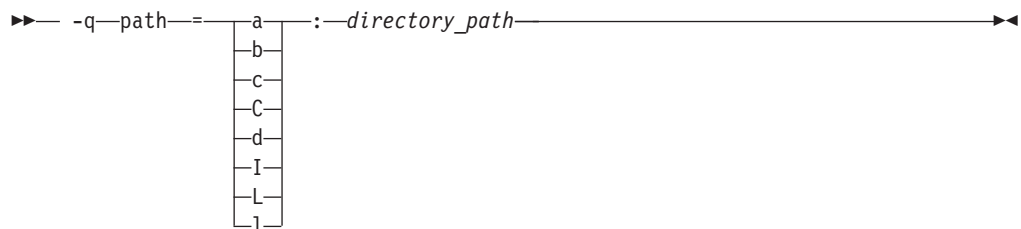
Defaults

By default, the compiler uses the paths for compiler components defined in the configuration file.

Parameters

directory_path
    The path to the directory where the alternate programs are located.

The following table shows the correspondence between **-qpath** parameters and the component executable names:

Parameter	Description	Executable name
a	Assembler	as
b	Low-level optimizer	xlCcode
c	Compiler front end	xlcentry, xlCentry
▶ C++ C	C++ compiler front end	xlCentry
d	Disassembler	dis

Parameter	Description	Executable name
I	High-level optimizer, compile step	ipa
L	High-level optimizer, link step	ipa
l	Linker	ld

Usage

The **-qpath** option overrides the **-F**, **-t**, and **-B** options.

Predefined macros

None.

Examples

To compile `myprogram.c` using a substitute **xlc** compiler in `/lib/tmp/mine/` enter:

```
xlc myprogram.c -qpath=c:/lib/tmp/mine/
```

To compile `myprogram.c` using a substitute linker in `/lib/tmp/mine/`, enter:

```
xlc myprogram.c -qpath=l:/lib/tmp/mine/
```

Related information
- "-B" on page 103
- "-F" on page 133
- "-t" on page 294

-qpdf1, -qpdf2
Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Tunes optimizations through *profile-directed feedback* (PDF), where results from sample program execution are used to improve optimization near conditional branches and in frequently executed code sections.

PDF is a two-step process. You first compile the application with **-qpdf1** and a minimum optimization level of **-O2**, with linking. You then run the resulting application with a typical data set. During the test run, profile data is written to a profile file. By default, the profile file is named ._pdf and is saved in the current working directory, or in the directory named by the *PDFDIR* environment variable, if it is set. You then recompile and link or relink the application with **-qpdf2** and an optimization level used for **-qpdf1**, which fine-tunes the optimizations applied according to the profile data collected during the program execution.

You can use old profiling information. In previous releases, when you modify the source file or compiler options and compile with **-qpdf2**, the compilation stops with an error. As of IBM XL C/C++ for Linux, V11.1, you see a list of warnings but compilation does not stop. However, using different compiler options between different stages of PDF does not give you any benefits for using PDF.

PDF is intended to be used after other debugging and tuning is finished, as one of the last steps before putting the application into production.

Syntax

```
                  ┌─nopdf2─────────────────────────────┐
                  ├─nopdf1─────────────────────────────┤
►►── -q ──────────┼─pdf1───────────────────────────────┼──────────────────►◄
                  │       ├──=──pdfname──=──file_path─┤ │
                  │       ├──=──exename─────────────────┤ │
                  │       ├──=──defname─────────────────┤ │
                  │       └──=──level──=──0──1──2───────┘ │
                  └─pdf2──────────────────────────────────┘
                          ├──=──pdfname──=──file_path─┤
                          ├──=──exename─────────────────┤
                          ├──=──defname─────────────────┤
                          └──=──level──=──0──1──2───────┘
```
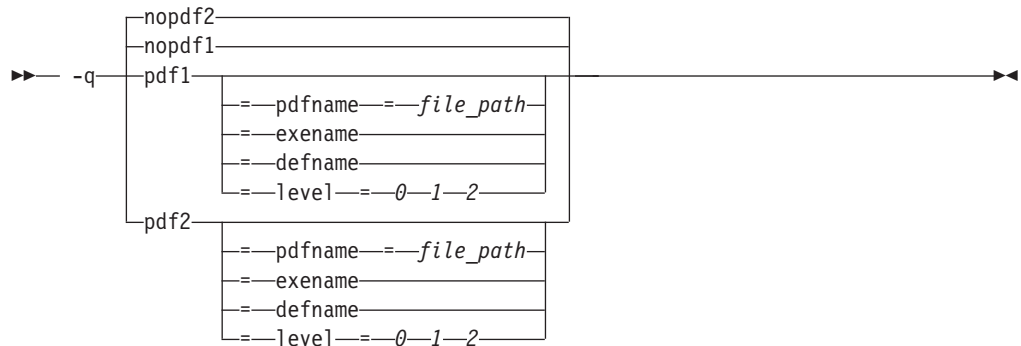
Defaults

-qnopdf1, -qnopdf2

Parameters

defname
Reverts the PDF file to its default file name.

exename
Generates the name of the PDF file based on what you specify with the **-o** option. For example, you can use **-qpdf1=exename -o foo foo.f** to generate a PDF file called **.foo_pdf**.

level=0 | 1 | 2
Supports multiple-pass profiling, block counter, call counter and extended value profiling. You can compile your application with **-qpdf1=level=0|1|2** to generate profiling data with different levels of optimization. Note that **-qpdf1=level=0** and **-qpdf1=level=1** support single-pass profiling, whereas **-qpdf=level=2** supports multiple-pass profiling. The following is a list of detailed descriptions for each level of optimization:

- **0** is the basic compiler instrumentation that generates lower overhead than **-qpdf1=level=1**.
- **1** is the default compiler instrumentation that is equivalent to **-qpdf1** in previous releases.
- **2** is a more aggressive compiler instrumentation. Cache-miss profiling is enabled on **pSeries**® in addition to basic block counter and value profiling performed at **-qpdf1=level=1**. This suboption is supported at all optimization levels where PDF is enabled. You can use **-qpdf1=level=2** to aggressively gather profile information, execute applications with typical input data, and use **-qpdf2** to optimize the executable.

pdfname= *file_path*

Specifies the path to the file that will hold the profile data. By default, the file name is ._pdf, and it is placed in the current working directory or in the directory named by the PDFDIR environment variable. You can use the **pdfname** suboption to allow you to do simultaneous runs of multiple executables using the same PDF directory. This is especially useful when tuning with PDF on dynamic libraries.

Usage

You must compile the main program with PDF for profiling information to be collected at run time.

If you do not want the optimized object files to be relinked during the second step, specify **-qpdf2 -qnoipa**.

If you want to specify an alternate path and file name for the profile file, use the **pdfname** suboption. Alternatively, you can use the *PDFDIR* environment variable to specify the absolute path name for the directory. To generate the name of the PDF files you specify, you can use the **exename** suboption. To revert the PDF file to its default name, use the **defname** suboption . Do not compile or run two different applications that use the same profiling directory at the same time, unless you have used the **pdfname** suboption to distinguish the sets of profiling information. For examples, see "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*.

When you run **-qprefetch=assistthread** to generate data prefetching assist threads, the compiler uses the delinquent load information to perform analysis and generate them. The delinquent load information can be gathered from dynamic profiling using **-qpdf1=level=2**. For more information, see -qprefetch.

You can also use the following option with **-qpdf1**:

-qshowpdf

Provides additional information, such as block and function call counts, to the profile file. See "-qshowpdf" on page 268 for more information.

For recommended procedures of using PDF, see "Using profile-directed feedback" in the *XL C/C++ Optimization and Programming Guide*.

The following utility programs, found in /opt/ibmcmp/vacpp/11.1/bin/, are available for managing the directory to which profile data is written:

cleanpdf

```
►►─cleanpdf─────────────────────────────────────────────────────►◄
              └─directory_path─┘
```

Removes all profiling information from the directory specified by *directory_path*; or if *pathname* is not specified, from the directory set by the PDFDIR environment variable; or if PDFDIR is not set, from the current directory. Removing profiling information reduces runtime overhead if you change the program and then go through the PDF process again.

Run **cleanpdf** only when you are finished with the PDF process for a particular application. Otherwise, if you want to resume using PDF with that application, you will need to recompile all of the files again with **-qpdf1**.

mergepdf

```
►►──mergepdf──┬─────────────┬──input──┬──────┬──-o──output──┬──────┬──┬──────┬──►◄
              │  ┌────────┐  │         └──────┘              └──-n──┘  └──-v──┘
              └──-r──scaling─┘
```
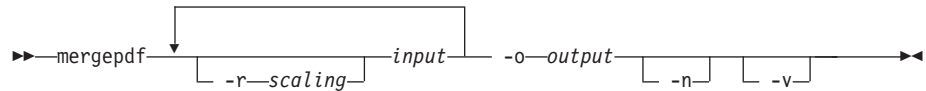
Merges two or more PDF records into a single PDF output record.

-r *scaling*
> Specifies the scaling ratio for the PDF record file. This value must be greater than zero and can be either an integer or floating point value. If not specified, a ratio of 1.0 is assumed.

input
> Specifies the name of a PDF input record file, or a directory that contains PDF record files.

-o *output*
> Specifies the name of the PDF output record file, or a directory to which the merged output will be written.

-n
> If specified, PDF record files are not normalized. If not specified, **mergepdf** normalizes records based on an internally-calculated ratio before applying any user-defined scaling factor.

-v
> Specifies verbose mode, and causes internal and user-specified scaling ratios to be displayed to standard output.

resetpdf

```
►►──resetpdf──┬────────────────┬──►◄
              └──directory_path─┘
```
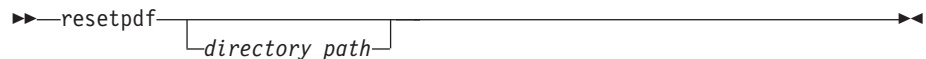
Same as **cleanpdf**, described above.

showpdf

```
►►──showpdf──┬────────────────┬──┬──────┬──┬───────────┬──►◄
             └──directory_path─┘  └──-f──┘  └──file_path─┘
```
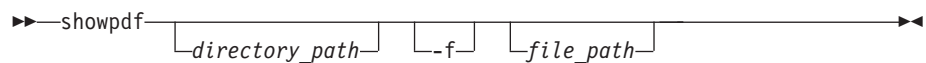
Displays the function call and block counts written to the profile file, specified by the **-f** option, during a program run. To use this command, you must first compile your application specifying both **-qpdf1** and **-qshowpdf** compiler options on the command line.

Predefined macros

None.

Examples

Here is a simple example:

```
// Compile all files with -qpdf1.
xlc -qpdf1 -O3 file1.c file2.c file3.c

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

// Recompile all files with -qpdf2.
xlc -qpdf2 -O3 file1.c file2.c file3.c

// The program should now run faster than
// without PDF if the sample data is typical.
```

Here is a more elaborate example.

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

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

// This file is not so important to optimize.
xlc -c file4.c

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

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

// No need to recompile the source of non-PDF object files (file4.c).
xlc -qpdf2 -O3 file1.c file2.c file3.c

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

Here is an example that bypasses recompiling the source with **-qpdf2**:

```
// Compile source with -qpdf1.
xlc -O3 -qpdf1 -c file.c

// Link in object file.
xlc -O3 -qpdf1 file.o

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

// Link in object file from qpdf1 pass.
// (Bypass source recompilation with -qpdf2.)
  xlc -O3 -qpdf2 file.o
```

Here is an example of using pdf1 and pdf2 objects:

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

// Link in object files.
xlc -qpdf1 -O3 file1.o file2.o

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

// Link in the mix of pdf1 and pdf2 objects.
  xlc -qpdf2 -O3 file1.o file2.o
```

Here is an example that creates PDF-optimized object files without relinking into an executable:

```
// Compile source with -qpdf1.
xlc -c -O3 -qpdf1 file1.c file2.c file3.c

// Link in object files.
xlc -O3 -qpdf1 file1.o file2.o file3.o

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

// Recompile the instrumented source files
xlc -c -O3 -qpdf2 -qnoipa file1.c file2.c file3.c
```

Here is an example that reduces possible runtime instrumentation overhead:

```
//Compile all files with -qpdf1=level=0.
xlc -qpdf1=level=0 -O3 file1.c file2.c file3.c

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

//Recompile all files with -qpdf2.
 xlc -qpdf2 -O3 file1.c file2.c file3.c

//The program should now run faster than
//without PDF if the sample data is typical.
```

Here is an example of multiple pass profiling:

```
//Compile all files with -qpdf1=level=2.
xlc -qpdf1=level=2 –O5 file1.c file2.c file3.c

//Run with one set of input data, the profiling information is recorded
//in ._pdf by default.
./a.out < sample.data

//Recompile all files with -qpdf1=level=2 again.
//The compiler will read the previous profile data,
//refine instrumentation and generate a new instrumented executable.

 xlc -qpdf1=level=2 -O5 file1.c file2.c file3.c

//Run it again, the profiling information is recorded in
//._pdf.1

./a.out < sample.data

//Recompile all files with -qpdf2

xlc -qpdf2 -O5 file1.c file2.c file3.c

//The program should now run faster than
//without PDF if the sample data is typical.
```

Here is an example that uses **-qpdf[1|2]=exename**:

```
//Compile all files with -qpdf1=exename.
xlc -qpdf1=exename –O5 -o final file1.c file2.c file3.c

//Run executable with sample input data.

./final < typical.data

//List the content of the directory.
 >ls -lrta
```

```
-rw-r--r-- 1 user staff 50 Dec 05 13:18 file1.c
-rw-r--r-- 1 user staff 50 Dec 05 13:18 file2.c
-rw-r--r-- 1 user staff 50 Dec 05 13:18 file3.c
-rwxr-xr-x 1 user staff 12243 Dec 05 17:00 final
-rwxr-Sr-- 1 user staff 762 Dec 05 17:03 .final_pdf

//Recompile all files with -qpdf2=exename.

xlc -qpdf2=exename —O5 -o final file1.c file2.c file3.c

//The program is now optimized using PDF information.
```

Related information

- "-qshowpdf" on page 268
- "-qipa" on page 176
- -qprefetch
- "-qreport" on page 254
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*
- "Runtime environment variables" on page 24
- "Profile-directed feedback" in the *XL C/C++ Optimization and Programming Guide*

-qphsinfo

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Reports the time taken in each compilation phase to standard output.

Syntax

```
►►── -q─┬─nophsinfo─┬──────────────────────────────────────────────►◄
        └─phsinfo───┘
```

Defaults

-qnophsinfo

Usage

The output takes the form *number1*/*number2* for each phase where *number1* represents the CPU time used by the compiler and *number2* represents the total of the compiler time and the time that the CPU spends handling system calls.

Predefined macros

None.

Examples

▶ **C** To compile `myprogram.c` and report the time taken for each phase of the compilation, enter:

```
xlc myprogram.c -qphsinfo
```

The output will look similar to:

```
C Init   - Phase Ends;  0.010/  0.040
IL Gen   - Phase Ends;  0.040/  0.070
W-TRANS  - Phase Ends;  0.000/  0.010
OPTIMIZ  - Phase Ends;  0.000/  0.000
REGALLO  - Phase Ends;  0.000/  0.000
AS       - Phase Ends;  0.000/  0.000
```

Compiling the same program with **-O4** gives:

```
C Init   - Phase Ends;  0.010/  0.040
IL Gen   - Phase Ends;  0.060/  0.070
IPA      - Phase Ends;  0.060/  0.070
IPA      - Phase Ends;  0.070/  0.110
W-TRANS  - Phase Ends;  0.060/  0.180
OPTIMIZ  - Phase Ends;  0.010/  0.010
REGALLO  - Phase Ends;  0.010/  0.020
AS       - Phase Ends;  0.000/  0.000
```

▶ **C++** To compile `myprogram.C` and report the time taken for each phase of the compilation, enter:

```
xlc++ myprogram.C -qphsinfo
```

The output will look similar to:

```
Front End - Phase Ends;  0.004/  0.005
W-TRANS  - Phase Ends;  0.010/  0.010
OPTIMIZ  - Phase Ends;  0.000/  0.000
REGALLO  - Phase Ends;  0.000/  0.000
AS       - Phase Ends;  0.000/  0.000
```

Compiling the same program with **-O4** gives:

```
Front End - Phase Ends;  0.004/  0.006
IPA      - Phase Ends;  0.040/  0.040
IPA      - Phase Ends;  0.220/  0.280
W-TRANS  - Phase Ends;  0.030/  0.110
OPTIMIZ  - Phase Ends;  0.030/  0.030
REGALLO  - Phase Ends;  0.010/  0.050
AS       - Phase Ends;  0.000/  0.000
```

-qpic

Category

Object code control

Pragma equivalent

None.

Purpose

Generates Position-Independent Code suitable for use in shared libraries.

Syntax

```
>>-- -q---+--nopic--+-------------------------------------><
          +--pic--+
                  |  +--small--+
                  +--=--+--large--+
```

Defaults

- **-qnopic** in 32-bit compilation mode.
- **-qpic=small** in 64-bit compilation mode.
- **-qpic=small** when the **-qmkshrobj** compiler option is specified.

Parameters

small
Instructs the compiler to assume that the size of the Global Offset Table is no larger than 64 Kb.

large
Allows the Global Offset Table to be larger than 64 Kb in size, allowing more addresses to be stored in the table. Code generated with this option is usually larger than that generated with **-qpic=small**.

Specifying **-qpic** without any suboptions is equivalent to **-qpic=small**.

Usage

When **-q64** is in effect, **-qpic** is enabled and cannot be disabled.

Predefined macros

None.

Examples

To compile a shared library libmylib.so, use the following commands:

```
xlc mylib.c -qpic=small -c -o mylib.o
xlc -qmkshrobj mylib -o libmylib.so.1
```

Related information
- "-q32, -q64" on page 87
- "-qmkshrobj" on page 224

-qppline

Category

Object code control

Pragma equivalent

None.

Purpose

When used in conjunction with the **-E** or **-P** options, enables or disables the generation of #line directives.

Syntax

```
►►── -q──┬─ppline────┬──────────────────────────────────────────────────────►◄
         └─noppline──┘
```

Defaults
- **-qnoppline** when **-P** is in effect
- **-qppline** when **-E** is in effect

Usage

The **-C** option has no effect without either the **-E** or the **-P** option. With the **-E** option, line directives are written to standard output. With the **-P** option, line directives are written to an output file.

Predefined macros

None.

Examples

To preprocess myprogram.c to write the output to myprogram.i, and generate #line directives:

```
xlc myprogram.c -P -qppline
```

Related information
- "-E" on page 127
- "-P" on page 232

-qprefetch

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Inserts prefetch instructions automatically where there are opportunities to improve code performance.

When **-qprefetch** is in effect, the compiler may insert prefetch instructions in compiled code. When **-qnoprefetch** is in effect, prefetch instructions are not inserted in compiled code.

Syntax

```
                                                         :
                         ┌─prefetch─┐     ┌──────────────────────────────┐
►►── -q ─────────────────┤          ├──┬──▼──────────────────────────────────────────►◄
                         └─noprefetch┘  │
                                        ├─=──assistthread──=──┬─SMT─┐
                                        │                     └─CMP─┘
                                        ├─=──noassistthread───
                                        │      ┌─noaggressive─┐
                                        └─=────┤              ├
                                               └─aggressive───┘
```
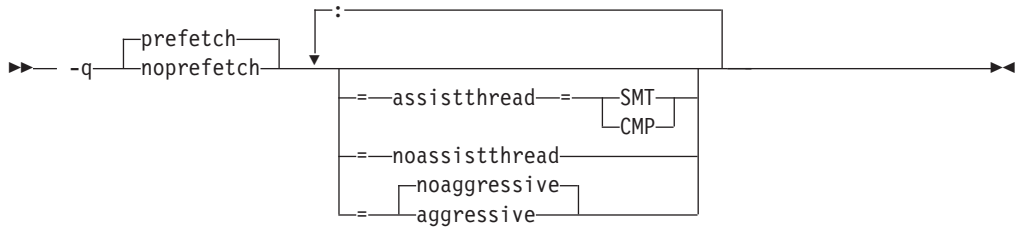
Defaults

- **-qprefetch**
- **-qprefetch=noassistthread**
- **-qprefetch=noassistthread:noaggressive**

Parameters

assistthread | noassistthread

When you work with applications that generate a high cache-miss rate, you can use **-qprefectch=assistthread** to exploit assist threads for data prefetching. This suboption guides the compiler to exploit assist threads at optimization level **-O3 -qhot** or higher. If you do not specify **-qprefetch=assistthread**, **-qprefetch=noassistthread** is implied.

aggressive | noaggressive

This suboption guides the compiler to generate aggressive data prefetching at optimization level **-O3 -qhot** or higher. If you do not specify **aggressive**, **-qprefetch=noaggressive** is implied.

CMP

For systems based on the chip multi-processor architecture (CMP), you can use **-qprefetch=assistthread=cmp**.

SMT

For systems based on the simultaneous multi-threading architecture (SMT), you can use **-qprefetch=assistthread=smt**.

Note: If you do not specify either CMP or SMT, the compiler uses the default setting based on your system architecture.

Usage

The **-qnoprefetch** option does not prevent built-in functions such as **__prefetch_by_stream** from generating prefetch instructions.

When you run **-qprefetch=assistthread**, the compiler uses the delinquent load information to perform analysis and generates prefetching assist threads. The delinquent load information can either be provided through the built-in **__mem_delay** function (const void *delinquent_load_address, const unsigned int delay_cycles), or gathered from dynamic profiling using **-qpdf1=level=2**.

When you use **-qpdf** to call **-qprefetch=assistthread**, you must use the traditional two-step PDF invocation:

1. Run **-qpdf1=level=2**
2. Run **-qpdf2 -qprefetch=assistthread**

Predefined macros

None.

Examples

Here is how you generate code using assist threads with __MEM_DELAY:

Initial code:

```
int y[64], x[1089], w[1024];

  void foo(void){
    int i, j;
    for (i = 0; i &l; 64; i++) {
      for (j = 0; j < 1024; j++) {

        /* what to prefetch? y[i]; inserted by the user */
        __mem_delay(&y[i], 10);
        y[i] = y[i] + x[i + j] * w[j];
        x[i + j + 1] = y[i] * 2;
    }
  }
}
```

Assist thread generated code:

```
void foo@clone(unsigned thread_id, unsigned version)

{ if (!1) goto lab_1;

/* version control to synchronize assist and main thread */
if (version == @2version0) goto lab_5;

goto lab_1;

lab_5:

@CIV1 = 0;

do { /* id=1 guarded */ /* ~2 */

if (!1) goto lab_3;

@CIV0 = 0;

do { /* id=2 guarded */ /* ~4 */

/* region = 0 */

/* __dcbt call generated to prefetch y[i] access */
__dcbt(((char *)&y + (4)*(@CIV1)))
@CIV0 = @CIV0 + 1;
} while ((unsigned) @CIV0 < 1024u); /* ~4 */

lab_3:
@CIV1 = @CIV1 + 1;
} while ((unsigned) @CIV1 < 64u); /* ~2 */

lab_1:

return;
}
```

Related information

- "-qarch" on page 95
- "-qhot" on page 155
- "-qpdf1, -qpdf2" on page 236
- "-qreport" on page 254
- "__mem_delay" on page 504

-qprint

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Enables or suppresses listings.

When **-qprint** is in effect, listings are enabled if they are requested by other compiler options that produce listings. When **-qnoprint** is in effect, all listings are suppressed, regardless of whether listing-producing options are specified.

Syntax

```
►►── -q───┬─print───┬──────────────────────────────────────── ►◄
          └─noprint─┘
```
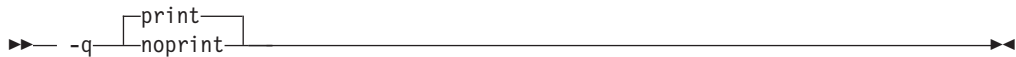
Defaults

-qprint

Usage

You can use **-qnoprint** to override all listing-producing options and equivalent pragmas, regardless of where they are specified. These options are:
- -qattr
- -qlist
- -qlistopt
- -qsource
- -qxref

Predefined macros

None.

Examples

To compile `myprogram.c` and suppress all listings, even if some files have **#pragma options source** and similar directives, enter:

```
xlc myprogram.c -qnoprint
```

-qpriority (C++ only)

Category

Object code control

Pragma equivalent

#pragma options priority, #pragma priority

Purpose

Specifies the priority level for the initialization of static objects.

The C++ standard requires that all global objects within the same translation unit be constructed from top to bottom, but it does not impose an ordering for objects declared in different translation units. The **-qpriority** option and **#pragma priority** directive allow you to impose a construction order for all static objects declared within the same load module. Destructors for these objects are run in reverse order during termination.

Syntax

Option syntax

▶▶── -q─priority──=──number──▶◀

Pragma syntax

▶▶──#──pragma──priority──(──number──)─────────────────────────────────────▶◀

Defaults

The default priority level is 65 535.

Parameters

number
    An integer literal in the range of 101 to 65 535. A lower value indicates a higher priority; a higher value indicates a lower priority. If you do not specify a *number*, the compiler assumes 65 535.

Usage

More than one **#pragma priority** can be specified within a translation unit. The priority value specified in one pragma applies to the constructions of all global objects declared after this pragma and before the next one. However, in order to be consistent with the Standard, priority values specified within the same translation unit must be strictly increasing. Objects with the same priority value are constructed in declaration order.

The effect of a **#pragma priority** exists only within one load module. Therefore, **#pragma priority** cannot be used to control the construction order of objects in different load modules.

Note: The C++ variable attribute `init_priority` can also be used to assign a priority level to a shared variable of class type. See "The init_priority variable attribute" in the *XL C/C++ Language Reference* for more information.

Examples

To compile the file `myprogram.C` to produce an object file `myprogram.o` so that objects within that file have an initialization priority of 2 000, enter:

```
 xlc++ myprogram.C -c -qpriority=2000
```

Related information
- "Initializing static objects in libraries" in the *XL C/C++ Optimization and Programming Guide*

-qprocimported, -qproclocal, -qprocunknown
Category

Optimization and tuning

Pragma equivalent

#pragma options proclocal, #pragma options procimported, #pragma options procunknown

Purpose

Marks functions as local, imported, or unknown in 64-bit compilations.

Local functions are statically bound with the functions that call them; smaller, faster code is generated for calls to such functions. You can use the **proclocal** option or pragma to name functions that the compiler can assume are local.

Imported functions are dynamically bound with a shared portion of a library. Code generated for calls to functions marked as imported may be larger, but is faster than the default code sequence generated for functions marked as unknown. You can use the **procimported** option or pragma to name functions that the compiler can assume are imported.

Unknown functions are resolved to either statically or dynamically bound objects during linking. You can use the **procunkown** option or pragma to name functions that the compiler can assume are unknown.

Syntax

```
►►── -q ──┬─ procunknown ──┬──────────────────────────────────────►◄
          ├─ proclocal ────┤
          └─ procimported ─┘
                              ┌─ : ─┐
                         └─ = ─▼─ function_name ─┘
```
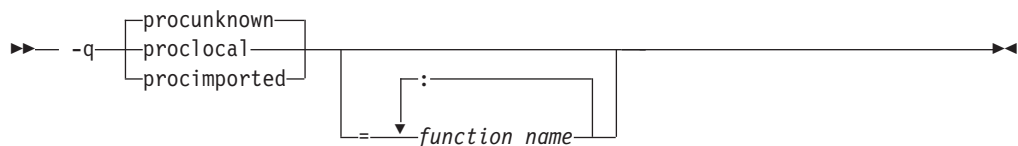
Defaults

-qprocunknown: The compiler assumes that all functions' definitions are unknown.

Parameters

function_name

> The name of a function that the compiler should assume is local, imported, or unknown (depending on the option specified). If you do not specify any *function_name*, the compiler assumes that *all* functions are local, imported, or unknown.

> ▶ `C++` Names must be specified using their mangled names. To obtain C++ mangled names, compile your source to object files only, using the **-c** compiler option, and use the **nm** operating system command on the resulting object file. (See also "Name mangling" in the *XL C/C++ Language Reference* for details on using the `extern "C"` linkage specifier on declarations to prevent name mangling.)

Usage

This option applies to 64-bit compilations only.

If any functions that are marked as local resolve to shared library functions, the linker will detect the error and issue warnings. If any of the functions that are marked as imported resolve to statically bound objects, the generated code may be larger and run more slowly than the default code sequence generated for unknown functions.

If you specify more than one of these options with no function names, the last option specified is used. If you specify the same function name on more than one option specification, the last one is used.

Predefined macros

None.

Examples

To compile `myprogram.c` along with the archive library `oldprogs.a` so that:

- Functions `fun` and `sun` are specified as local
- Functions `moon` and `stars` are specified as imported
- Function `venus` is specified as unknown

use the following command:

```
xlc myprogram.c oldprogs.a -qprolocal=fun(int):sun()
  -qprocimported=moon():stars(float) -qprocunknown=venus()
```

If the following example, in which a function marked as local instead resolves to a shared library function, is compiled with **-qproclocal**:

```
int main(void)
{
    printf("Just in function foo1()\n");
    printf("Just in function foo1()\n");
}
```

a linker error will result. To correct this problem, you should explicitly mark the called routine as being imported from a shared object. In this case, you would recompile the source file and explicitly mark `printf` as imported by compiling with `-qproclocal -qprocimported=printf`.

Related information
- "-qdataimported, -qdatalocal, -qtocdata" on page 121

-qproto (C only)
Category

Object code control

Pragma equivalent

#pragma options [no]proto

Purpose

Specifies the linkage conventions for passing floating-point arguments to functions that have not been prototyped.

When **proto** is in effect, the compiler assumes that the arguments in function calls are the same types as the corresponding parameters of the function definition, even if the function has not been prototyped. By asserting that an unprototyped function actually expects a floating-point argument if it is called with one, you allow the compiler to pass floating-point arguments in floating-point registers exclusively. When **noproto** is in effect, the compiler does not make this assumption, and must pass floating-point parameters in floating-point and general purpose registers.

Syntax

```
            ┌─noproto─┐
►►── -q ────┼─proto───┤──────────────────────────────────────────►◄
```

Defaults

-qnoproto

Usage

This option is only valid when the compiler allows unprototyped functions; that is, with the **cc** or **xlc** invocation command, or with the **-qlanglvl** option set to **classic | extended | extc89 | extc99**.

Predefined macros

None.

Examples

To compile `my_c_program.c` to allow the compiler to use the standard linkage conventions for floating-point parameters, even when functions are not prototyped, enter:

```
xlc my_c_program.c -qproto
```

-r

Category

Object code control

Pragma equivalent

None.

Purpose

Produces a nonexecutable output file to use as an input file in another ld command call. This file may also contain unresolved symbols.

Syntax

►►— -r ───►◄

Defaults

Not applicable.

Usage

A file produced with this flag is expected to be used as an input file in another compiler invocation or ld command call.

Predefined macros

None.

Examples

To compile `myprogram.c` and `myprog2.c` into a single object file `mytest.o`, enter:

```
xlc myprogram.c myprog2.c -r -o mytest.o
```

-R

Category

Linking

Pragma equivalent

None.

Purpose

At link time, writes search paths for shared libraries into the executable, so that these directories are searched at program run time for any required shared libraries.

Syntax

►►── -R──*directory_path*──►◄

Defaults

The default is to include only the standard directories. See the compiler configuration file for the directories that are set by default.

Usage

If the **-R***directory_path* option is specified both in the configuration file and on the command line, the paths specified in the configuration file are searched first at run time.

The **-R** compiler option is cumulative. Subsequent occurrences of **-R** on the command line do not replace, but add to, any directory paths specified by earlier occurrences of **-R**.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the directory `/usr/tmp/old` is searched at run time along with standard directories for the dynamic library `libspfiles.so`, enter:

```
xlc myprogram.c -lspfiles -R/usr/tmp/old
```

Related information
- "-L" on page 187

-qreport
Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Produces listing files that show how sections of code have been optimized.

A listing file is generated with a .lst suffix for each source file named on the command line. When used with an option that enables automatic parallelization or vectorization, the listing file shows a pseudo-C code listing and a summary of how program loops are parallelized or optimized. The report also includes diagnostic information to show why specific loops could not be parallelized or vectorized. For instance, when **-qreport** is used with **-qsimd=auto**, messages are provided to identify non-stride-one references that can prevent loop vectorization.

The compiler also reports the number of streams created for a given loop, which include both load and store streams. This information is included in the `Loop Transformation` section of the listing file. You can use this information to understand your application code and to tune your code for better performance. For example, you can distribute a loop which has more streams than the number supported by the underlying architecture. POWER4 and POWER5 support load stream prefetch and POWER6 supports both load and store stream prefetch.

Syntax

```
►►── -q──┬─noreport─┬──────────────────────────────────────────────►◄
         └─report───┘
```

Defaults

-qnoreport

Usage

For **-qreport** to generate a loop transformation listing, you must also specify one of the following on the command line:
- **-qsimd=auto**
- **-qsmp**
- **-qhot=level=2** and **-qsmp**
- **-O5**
- **-qipa=level=2**

For **-qreport** to generate PDF information in the listing, you must specify the following option in the command line:
- **-qpdf2 -qreport**

For **-qreport** to generate a parallel transformation listing or parallel performance messages, you must also specify one of the following options on the command line:
- **-qsmp**
- **-O5**
- **-qipa=level=2**

To generate data reorganization information, specify the optimization level **-qipa=level=2** or **-O5** together with **-qreport**. Reorganizations include array splitting, array transposing, memory allocation merging, array interleaving, and array coalescing.

To generate information about data prefetch insertion locations, use the optimization level of **-qhot**, or any other option that implies **-qhot** together with **-qreport**. This information appears in the `LOOP TRANSFORMATION SECTION` of the listing file. In addition, when you use **-qprefetch=assistthread** to generate prefetching assist threads, the message: `Assist thread for data prefetching was generated` also appears in the `LOOP TRANSFORMATION SECTION` of the listing file.

To generate a list of aggressive loop transformations and parallelizations performed on loop nests in the `LOOP TRANSFORMATION SECTION` of the listing file, use the optimization level of **-qhot=level=2** and **-qsmp** together with **-qreport**.

The pseudo-C code listing is not intended to be compilable. Do not include any of the pseudo-C code in your program, and do not explicitly call any of the internal routines whose names may appear in the pseudo-C code listing.

Predefined macros

None.

Examples

To compile `myprogram.c` so the compiler listing includes a report showing how loops are optimized, enter:

```
xlc -qhot -O3 -qreport myprogram.c
```

To compile `myprogram.c` so the compiler listing also includes a report showing how parallelized loops are transformed, enter:

```
xlc_r -qhot -qsmp -qreport myprogram.f
```

Related information
- "-qhot" on page 155
- "-qsimd" on page 269
- "-qipa" on page 176
- "-qsmp" on page 272
- "-qoptdebug" on page 230
- "-qprefetch" on page 245
- "Using -qoptdebug to help debug optimized programs" in the *XL C/C++ Optimization and Programming Guide*

-qreserved_reg
Category

Object code control

Pragma equivalent

None.

Purpose

Indicates that the given list of registers cannot be used during the compilation except as a stack pointer, frame pointer or in some other fixed role.

You should use this option in modules that are required to work with other modules that use global register variables or hand-written assembler code.

Syntax

```
►►── -q─reserved_reg─=─┬─register_name─┬──────────────►◄
                       └──────:◄──────┘
```

Defaults

Not applicable.

Parameters

register_name

A valid register name on the target platform. Valid registers are:

r0 to r31

General purpose registers

f0 to f31

Floating-point registers

v0 to v31

Vector registers (on selected processors only)

Usage

-qreserved_reg is cumulative, for example, specifying **-qreserved_reg=r14** and **-qreserved_reg=r15** is equivalent to specifying **-qreserved_reg=r14:r15**.

Duplicate register names are ignored.

Predefined macros

None.

Examples

To specify that `myprogram.c` reserves the general purpose registers r3 and r4, enter:

```
xlc myprogram.c -qreserved_reg=r3:r4
```

Related information
- "Variables in specified registers" in the *XL C/C++ Language Reference*

-qrestrict (C only)
Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Indicates to the compiler that no other pointers can access the same memory that has been addressed by function parameter pointers.

Specifying this option is equivalent to adding the `restrict` keyword to the pointer parameters within the specified functions, except that you do not need to modify the source file.

Syntax

```
        ┌─norestrict─┐
►►──-q──┤            ├─────────────────────────────►◄
        └─restrict───┘
              ┌──────:──────┐
              │      ▼       │
              └─=─────function_name──┘
```
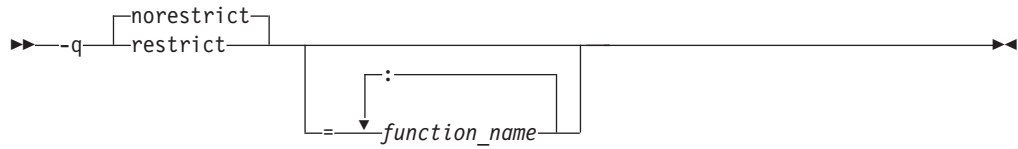
Defaults

-qnorestrict. It means no function pointer parameters are restricted, unless you specify the **restrict** attribute in the source.

Usage

If you do not specify the *function_name*, pointer parameters in all functions are treated as **restrict**. Otherwise, only those pointer parameters in the listed functions are treated as **restrict**.

function_name is a colon-separated list.

Using this option can improve the performance of your application, but incorrectly asserting this pointer restriction might cause the compiler to generate incorrect code based on the false assumption. If the application works correctly when recompiled without **-qrestrict**, the assertion might be false. In this case, this option should not be used.

Notes:
- Using **-qnokeyword=restrict** has no impact on the **-qrestrict** option.
- If you use both **-qalias=norestrict** and the **-qrestrict** option, the last one specified is used.

Predefined macros

None.

Examples

To compile myprogram.c, instructing the compiler to restrict the pointer access, enter:

```
xlc -qrestrict myprogram.c
```

Related information
- The restrict type qualifier in the *XL C/C++ Language Reference*.
- Keywords in the *XL C/C++ Language Reference*.
- -qkeyword
- -qalias

-qro

Category

Object code control

Pragma equivalent

#pragma options ro, #pragma strings

Purpose

Specifies the storage type for string literals.

When **ro** or **strings=readonly** is in effect, strings are placed in read-only storage. When **noro** or **strings=writeable** is in effect, strings are placed in read/write storage.

Syntax

Option syntax

```
►►── -q──┬─ro───┬──────────────────────────────────────────────────►◄
         └─noro─┘
```

Pragma syntax

```
►►──#──pragma──strings──(──┬─readonly──┬──)──────────────────────►◄
                           └─writeable─┘
```

Defaults

▷ **C** Strings are read-only for all invocation commands except **cc**. If the **cc** invocation command is used, strings are writeable.

▷ **C++** Strings are read-only.

Parameters

readonly (pragma only)
String literals are to be placed in read-only memory.

writeable (pragma only)
String literals are to be placed in read-write memory.

Usage

Placing string literals in read-only memory can improve runtime performance and save storage. However, code that attempts to modify a read-only string literal may generate a memory error.

The pragmas must appear before any source statements in a file.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the storage type is writable, enter:

```
xlc myprogram.c -qnoro
```

Related information

- "-qro" on page 258
- "-qroconst"

-qroconst

Category

Object code control

Pragma equivalent

#pragma options [no]roconst

Purpose

Specifies the storage location for constant values.

When **roconst** is in effect, constants are placed in read-only storage. When **noroconst** is in effect, constants are placed in read/write storage.

Syntax

```
                  ┌─roconst───┐
►►─── -q──────┴─noroconst─┴──────────────────────────────────────────◄◄
```

Defaults

- ▶ C ◀ **-qroconst** for all compiler invocations except **cc** and its derivatives. **-qnoroconst** for the **cc** invocation and its derivatives.

- ▶ C++ ◀ **-qroconst**

Usage

Placing constant values in read-only memory can improve runtime performance, save storage, and provide shared access. However, code that attempts to modify a read-only constant value generates a memory error.

"Constant" in the context of the **-qroconst** option refers to variables that are qualified by const, including const-qualified characters, integers, floats, enumerations, structures, unions, and arrays. The following constructs are not affected by this option:

- Variables qualified with volatile and aggregates (such as a structure or a union) that contain volatile variables
- Pointers and complex aggregates containing pointer members
- Automatic and static types with block scope
- Uninitialized types
- Regular structures with all members qualified by const
- Initializers that are addresses, or initializers that are cast to non-address values

The **-qroconst** option does not imply the **-qro** option. Both options must be specified if you wish to specify storage characteristics of both string literals (**-qro**) and constant values (**-qroconst**).

Predefined macros

None.

Related information
- "-qro" on page 258

-qrtti (C++ only)
Category

Object code control

Pragma equivalent

#pragma options rtti

Purpose

Generates runtime type identification (RTTI) information for exception handling and for use by the `typeid` and `dynamic_cast` operators.

Syntax

```
►►── -q──┬─rtti──┬────────────────────────────────────────────────►◄
         └─nortti─┘
```

Defaults

-qrtti

Usage

For improved runtime performance, suppress RTTI information generation with the **-qnortti** setting.

You should be aware of the following effects when specifying the **-qrtti** compiler option:
- Contents of the virtual function table will be different when **-qrtti** is specified.
- When linking objects together, all corresponding source files must be compiled with the correct **-qrtti** option specified.
- If you compile a library with mixed objects (**-qrtti** specified for some objects, **-qnortti** specified for others), you may get an undefined symbol error.

Predefined macros
- __RTTI_ALL__ is defined to 1 when **-qrtti** is in effect; otherwise, it is undefined.
- __NO_RTTI__ is defined to 1 when **-qnortti** is in effect; otherwise, it is undefined.

Related information
- "-qeh (C++ only)" on page 129

-s

Category

Object code control

Pragma equivalent

None.

Purpose

Strips the symbol table, line number information, and relocation information from the output file.

This command is equivalent to the operating system **strip** command.

Syntax

►►── -s ───►◄

Defaults

The symbol table, line number information, and relocation information are included in the output file.

Usage

Specifying **-s** saves space, but limits the usefulness of traditional debug programs when you are generating debug information using options such as **-g**.

Predefined macros

None.

Related information
- "-g" on page 149

-S

Category

Output control

Pragma equivalent

None.

Purpose

Generates an assembler language file for each source file.

The resulting file has an .s suffix and can be assembled to produce object .o files or an executable file (a.out).

Syntax

```
►►─ -S ──────────────────────────────────────────────────────────►◄
```

Defaults

Not applicable.

Usage

You can invoke the assembler with any compiler invocation command. For example,

```
xlc myprogram.s
```

will invoke the assembler, and if successful, the linker to create an executable file, a.out.

If you specify **-S** with **-E** or **-P**, **-E** or **-P** takes precedence. Order of precedence holds regardless of the order in which they were specified on the command line.

You can use the **-o** option to specify the name of the file produced only if no more than one source file is supplied. For example, the following is *not* valid:

```
xlc myprogram1.c myprogram2.c -o -S
```

Predefined macros

None.

Examples

To compile myprogram.c to produce an assembler language file myprogram.s, enter:

```
xlc myprogram.c -S
```

To assemble this program to produce an object file myprogram.o, enter:

```
xlc myprogram.s -c
```

To compile myprogram.c to produce an assembler language file asmprogram.s, enter:

```
xlc myprogram.c -S -o asmprogram.s
```

Related information
- "-E" on page 127
- "-P" on page 232

-qsaveopt
Category

Object code control

Pragma equivalent

None.

Purpose

Saves the command-line options used for compiling a source file, the user's configuration file name and the options specified in the configuration files, the version and level of each compiler component invoked during compilation, and other information to the corresponding object file.

Syntax

```
►►── -q──┬─nosaveopt─┬──────────────────────────────────────────────►◄
         └─saveopt───┘
```

Defaults

-qnosaveopt

Usage

This option has effect only when compiling to an object (.o) file (that is, using the **-c** option). Though each object might contain multiple compilation units, only one copy of the command-line options is saved. Compiler options specified with pragma directives are ignored.

Command-line compiler options information is copied as a string into the object file, using the following format:

```
►►──@(#)──opt──┬─f─┬──invocation──options────────────────────────────►◄
               ├─c─┤
               └─C─┘
```

```
►►──@(#)──cfg────config_file_options_list────────────────────────────►◄
```

```
►►──@(#)──evn────env_var_definition───────────────────────────────────►◄
```

where:

f Signifies a Fortran language compilation.

c Signifies a C language compilation.

C Signifies a C++ language compilation.

invocation
   Shows the command used for the compilation, for example, **xlc**.

options The list of command line options specified on the command line, with individual options separated by space.

config_file_options_list
   The list of options specified by the **options** attribute in all configuration files that take effect in the compilation, separated by space.

env_var_definition
   The environment variables that are used by the compiler. Currently only **XLC_USR_CONFIG** is listed.

   Note: You can always use this option, but the corresponding information is only generated when the environment variable **XLC_USR_CONFIG** is set.

For more information about the environment variable **XLC_USR_CONFIG**, see Compile-time and link-time environment variables.

Note: The string of the command-line options is truncated after 64k bytes.

Compiler version and release information, as well as the version and level of each component invoked during compilation, are also saved to the object file in the format:
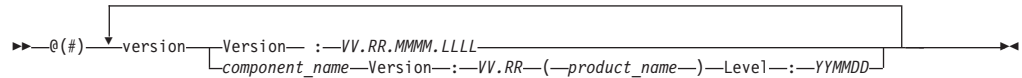
```
►►—@(#)—┬─version─┬─Version— :—VV.RR.MMMM.LLLL───────────────────────────┬──►◄
         │         └─component_name—Version—:—VV.RR—(—product_name—)—Level—:—YYMMDD─┘
```

where:
V	Represents the version.
R	Represents the release.
M	Represents the modification.
L	Represents the level.

component_name
> Specifies the components that were invoked for this compilation, such as the low-level optimizer.

product_name
> Indicates the product to which the component belongs (for example, C/C++ or Fortran).

YYMMDD
> Represents the year, month, and date of the installed update. If the update installed is at the base level, the level is displayed as BASE.

If you want to simply output this information to standard output without writing it to the object file, use the **-qversion** option.

Predefined macros

None.

Examples

Compile t.c with the following command:
```
xlc t.c -c -qsaveopt -qhot
```

Issuing the **strings -a** command on the resulting t.o object file produces information similar to the following:
```
opt c /opt/ibmcmp/vac/11.1/bin/xlc t.f -c -qsaveopt -qhot
cfg -qlanglvl=extc99 -qcpluscmt -qkeyword=inline -qalias=ansi -D_AIX -D_AIX32
 -D_AIX41 -D_AIX43 -D_AIX50 -D_AIX51 -D_AIX52 -D_AIX53 -D_IBMR2 -D_POWER
version IBM XL C/C++ for Linux, V11.1
version Version: 11.01.0000.0000
version Driver Version: 11.01(C/C++) Level: YYMMDD
version Front End  Version: 11.01(C/C++)  Level: YYMMDD
version C Front End Version : 11.01(C/C++)  Level: YYMMDD
version High Level Optimizer Version: 11.01(C) and 13.01(Fortran)  Level: YYMMDD
version Low Level Optimizer  Version: 11.01(C) and 13.01(Fortran) Level: YYMMDD
```

In the first line, c identifies the source used as C, /opt/ibmcmp/vacpp/11.1/bin/xlc shows the invocation command used, and **-qhot -qsaveopt** shows the compilation options.

The remaining lines list each compiler component invoked during compilation, and its version and level. Components that are shared by multiple products may show more than one version number. Level numbers shown may change depending on the updates you have installed on your system.

Related information

- "-qversion" on page 317

-qshowinc

Category

Listings, messages, and compiler information

Pragma equivalent

#pragma options [no]showinc

Purpose

When used with **-qsource** option to generate a listing file, selectively shows user or system header files in the source section of the listing file.

Syntax
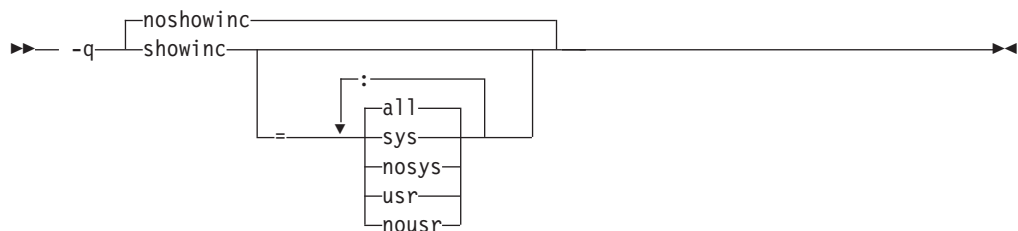
Defaults

-qnoshowinc: Header files included in source files are not shown in the source listing.

Parameters

all Shows both user and system include files in the program source listing.

sys
>Shows system include files (that is, files included with the `#include <filename>` preprocessor directive) in the program source listing.

usr
>Shows user include files (that is, files included with the `#include "filename"` preprocessor directive or with **-qinclude**) in the program source listing.

Specifying **showinc** with no suboptions is equivalent to **-qshowinc=sys : usr** and **-qshowinc=all**. Specifying **noshowinc** is equivalent to **-qshowinc=nosys : nousr**.

Usage

This option has effect only when the **-qlist** or **-qsource** compiler options is in effect.

Predefined macros

None.

Examples

To compile `myprogram.c` so that all included files appear in the source listing, enter:

```
xlc myprogram.c -qsource -qshowinc
```

Related information
- "-qsource" on page 276

-qshowmacros
Category

"Output control" on page 67

Pragma equivalent

None

Purpose

Emits macro definitions to preprocessed output.

Emitting macros to preprocessed output can help determine functionality available in the compiler. The macro listing may prove useful for debugging complex macro expansions, as well.

Syntax

```
                 ┌─noshowmacros─────────────────────┐
►►── -q──┴─showmacros─┴──────────────────────────────►◄
                 │        ┌──────:──────┐         │
                 └─=─┬──▼──┬─all──┬───────┘
                        ├─nopre─┤
                        └─pre───┘
```
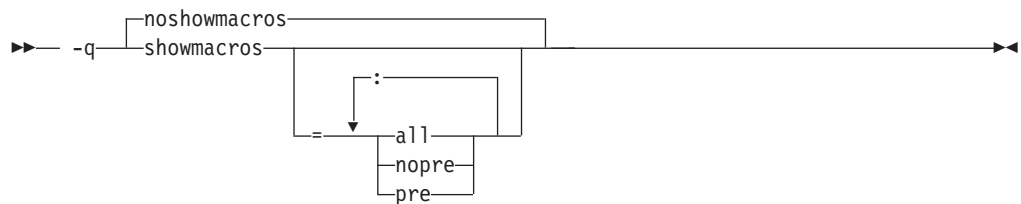
Defaults

-qnoshowmacros

Parameters

all Emits all macro definitions to preprocessed output. This is the same as specifying **-qshowmacros**.

pre | **nopre**
   pre emits only predefined macro definitions to preprocessed output. **nopre** suppresses appending these definitions.

Usage

Note the following when using this option:

- This option has no effect unless preprocessed output is generated; for example, by using the **-E** or **-P** options.
- If a macro is defined and subsequently undefined before compilation ends, this macro will not be included in the preprocessed output.
- Only macros defined internally by the preprocessor are considered predefined; all other macros are considered as user-defined.

Related information

- "-E" on page 127
- "-P" on page 232

-qshowpdf

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

When used with **-qpdf1** and a minimum optimization level of **-O2** at compile and link steps, inserts additional profiling information into the compiled application to collect call and block counts for all procedures in the application.

Syntax

```
              ┌─noshowpdf─┐
►►── -q───────┴─showpdf───┴──────────────────────────────────────►◄
```

Usage

After you run your application with training data, the call and block counts are recorded in the profile file (by default, this is named ._pdf). You can retrieve the contents of the profile file with the **showpdf** utility, described in "-qpdf1, -qpdf2" on page 236.

For procedures and examples of using **-qshowdpf** and **showpdf**, see "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*.

Predefined macros

None.

Related information

- "-qpdf1, -qpdf2" on page 236
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*

-qsimd

Category

Optimization and tuning

Pragma equivalent

#pragma nosimd

Purpose

Controls whether the compiler can automatically take advantage of vector instructions for processors that support them.

These instructions can offer higher performance when used with algorithmic-intensive tasks such as multimedia applications.

Syntax

```
►►─── -q─simd──=─┬─noauto─┬─────────────────────────────────────►◄
                 └─auto───┘
```

Defaults

-qsimd=noauto

Usage

The **-qsimd=auto** option enables automatic generation of vector instructions for processors that support them. It replaces the **-qenablevmx** option, which has been deprecated.

When **-qsimd=auto** is in effect, the compiler converts certain operations that are performed in a loop on successive elements of an array into vector instructions. These instructions calculate several results at one time, which is faster than calculating each result sequentially. Applying this option is useful for applications with significant image processing demands.

The **-qsimd=noauto** option disables the conversion of loop array operations into vector instructions. Finer control can be achieved by using **-qstrict=ieeefp**, **-qstrict=operationprecision**, and **-qstrict=vectorprecision**. For details, see "-qstrict" on page 286.

Note: Using vector instructions to calculate several results at one time might delay or even miss detection of floating point exceptions on some architectures. If detecting exceptions is important, do not use **-qsimd=auto**.

The following rules apply when you use the **-qsimd** option:
* Specifying the deprecated **-qenablevmx** option has the same effect as specifying **-qsimd=auto**. The compiler does not issue any warning for this.
* Specifying **-qsimd** without any suboption has the same effect as **-qsimd=auto**.
* This option is available only when you set **-qarch** to a target architecture that supports vector instructions.

- Specify **-qsimd=auto** only when your processor architecture supports vector processing.
- If you specify **-qsimd=auto** to enable IPA at the compile time but specify **-simd=noauto** at the link time, the compiler automatically sets **-qsimd=auto** and sets an appropriate value for **-qarch** to match the architecture specified at the compile time.

Predefined macros

None.

Examples

The following example shows the usage of **#pragma nosimd** to disable **-qsimd=auto** for a specific `for` loop:

```
...
#pragma nosimd
for (i=1; i<1000; i++) {
    /* program code */
}

...
```

Related information
- "-qarch" on page 95
- "-qstrict" on page 286

-qskipsrc

Category

"Listings, messages, and compiler information" on page 76

Pragma equivalent

None.

Purpose

When a listing file is generated using the **-qsource** option, **-qskipsrc** can be used to determine whether the source statements skipped by the compiler are shown in the source section of the listing file. Alternatively, the **-qskipsrc=hide** option is used to hide the source statements skipped by the compiler.

Syntax

```
             ┌─show─┐
►►── -q─skipsrc──=──┤      ├────────────────────────────────────◄
             └─hide─┘
```

Defaults
- -qskipsrc=show

Parameters

show | **hide**

> When **show** is in effect, the compiler will display all source statements in the listing. This will result in both true and false paths of the preprocessing directives to be shown.
>
> On the contrary, when **hide** is enabled, all source statements that the compiler skipped will be omitted.

Usage

In general, the **-qskipsrc** option does not control whether the source section is included in the listing file, it only does so when the **-qsource** option is in effect.

To display all source statements in the listing (default option):

```
xlc myprogram.c -qsource -qskipsrc=show
```

To omit source statements skipped by the compiler:

```
xlc myprogram.c -qsource -qskipsrc=hide
```

Predefined macros

None.

Related information
- "-qsource" on page 276
- "-qshowinc" on page 266
- "-qsrcmsg (C only)" on page 280

-qsmallstack

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Reduces the size of the stack frame.

Syntax

```
►►─ -q──┬─nosmallstack─┬──────────────────────────────────────►◄
         └─smallstack───┘
```

Defaults

-qnosmallstack

Usage

Programs that allocate large amounts of data to the stack, such as threaded programs, may result in stack overflows. This option can reduce the size of the stack frame to help avoid overflows.

This option is only valid when used together with IPA (**-qipa**, **-O4**, **-O5** compiler options).

Specifying this option may adversely affect program performance.

Predefined macros

None.

Examples

To compile myprogram.c to use a small stack frame, enter:

```
xlc myprogram.c -qipa -qsmallstack
```

Related information
- "-g" on page 149
- "-qipa" on page 176
- "-O, -qoptimize" on page 227

-qsmp

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Enables parallelization of program code.

Syntax

```
                    ┌─nosmp─────────────────────────────────────────────┐
►►─── -q ───┬─smp───┴───────────────────────────────────────────────────┴───►◄
                    │                     ┌─:──────────────────────────┐
                    │                     │  ┌─nostackcheck─────────┐  │
                    │                     │  ├─ostls────────────────┤  │
                    │                     │  ├─opt─────────────────┤  │
                    │                     │  ├─norec_locks─────────┤  │
                    │                     │  ├─noomp───────────────┤  │
                    │                     │  ├─nonested_par────────┤  │
                    │                     │  ├─explicit────────────┤  │
                    │                     ▼  ├─auto────────────────┤  │
                    └─=─┬───────────────────┼─omp─────────────────┼──┘
                                            ├─noostls──────────────┤
                                            ├─nested_par───────────┤
                                            ├─noauto───────────────┤
                                            ├─noexplicit───────────┤
                                            ├─noopt────────────────┤
                                            ├─rec_locks────────────┤
                                            │              ┌─runtime──┐       │
                                            ├─schedule──=──┼─auto─────┤       │
                                            │              ├─affinity─┐       │
                                            │              ├─dynamic──┤ ┌─=─n─┐│
                                            │              ├─guided───┤       │
                                            │              └─static───┘       │
                                            ├─stackcheck───────────┤
                                            └─threshold──┬─────────┬┘
                                                         └─=──n────┘
```
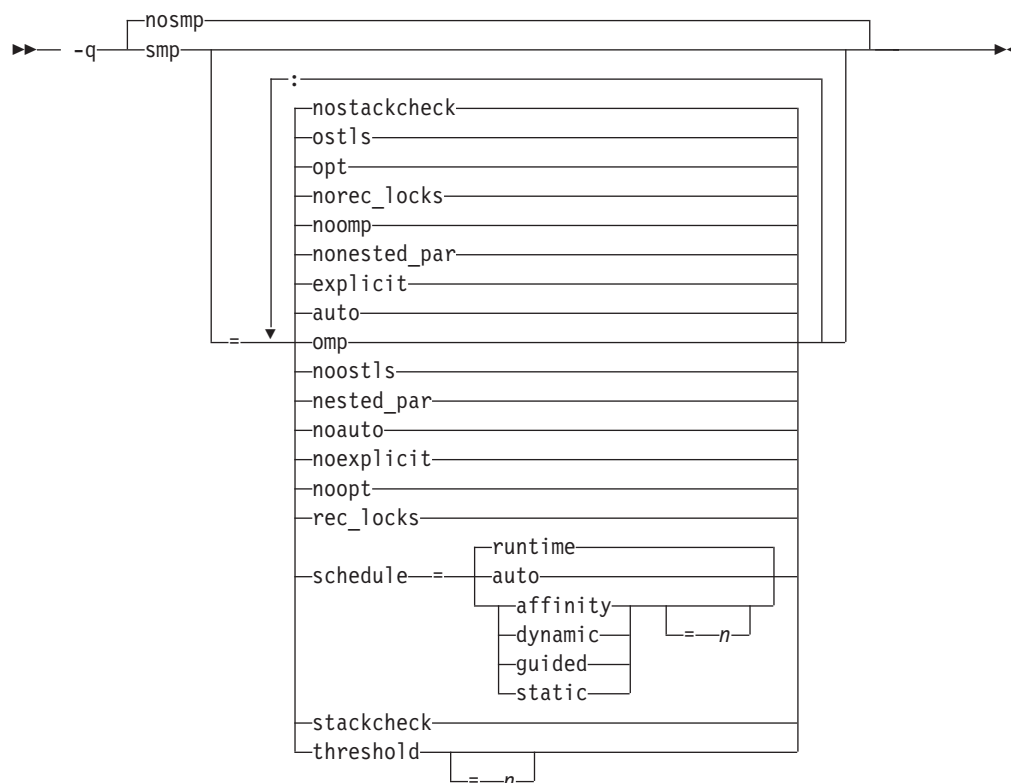
Defaults

-qnosmp. Code is produced for a uniprocessor machine.

Parameters

auto | noauto
> Enables or disables automatic parallelization and optimization of program code. When **noauto** is in effect, only program code explicitly parallelized with OpenMP directives is optimized. **noauto** is implied if you specify **-qsmp=omp** or **-qsmp=noopt**.

explicit | noexplicit
> Enables or disables directives controlling explicit parallelization of loops.

nested_par | nonested_par
> By default, the compiler serializes a nested parallel construct. When **nested_par** is in effect, the compiler parallelizes prescriptive nested parallel constructs. This includes not only the loop constructs that are nested within a scoping unit but also parallel constructs in subprograms that are referenced (directly or indirectly) from within other parallel constructs. Note that this suboption has no effect on loops that are automatically parallelized. In this case, at most one loop in a loop nest (in a scoping unit) will be parallelized. **nested_par** does not provide true nested parallelism because it does not cause new team of threads to be created for nested parallel regions. Instead, threads that are currently available are reused.
>
> This suboption should be used with caution. Depending on the number of threads available and the amount of work in an outer loop, inner loops could be executed sequentially even if this option is in effect. Parallelization overhead may not necessarily be offset by program performance gains.

Note:
- The implementation of the **nested_par** suboption does not comply with the OpenMP API.
- If you specify this suboption, the runtime library uses the same threads for the nested constructs that it used for the enclosing constructs.

omp | noomp

Enforces or relaxes strict compliance to the OpenMP standard. When **noomp** is in effect, **auto** is implied. When **omp** is in effect, **noauto** is implied and only OpenMP parallelization directives are recognized. The compiler issues warning messages if your code contains any language constructs that do not conform to the OpenMP API.

opt | noopt

Enables or disables optimization of parallelized program code. When **noopt** is in effect, the compiler will do the smallest amount of optimization that is required to parallelize the code. This is useful for debugging because **-qsmp** enables the **-O2** and **-qhot** options by default, which may result in the movement of some variables into registers that are inaccessible to the debugger. However, if the **-qsmp=noopt** and **-g** options are specified, these variables will remain visible to the debugger.

ostls | noostls

Enables Thread Local Storage (TLS) provided by the operating system to be used for **threadprivate** data. You can use the **noostls** suboption to enable the non-TLS for **threadprivate**. The **noostls** suboption is provided for backward compatibility.

Note: If you want to use this suboption, your operating system must support TLS to implement OpenMP **threadprivate** data. Use **noostls** to disable OS level TLS if your operating system does not support it.

rec_locks | norec_locks

Determines whether recursive locks are used. When **rec_locks** is in effect, nested critical sections will not cause a deadlock. Note that the **rec_locks** suboption specifies behavior for critical constructs that is inconsistent with the OpenMP API.

schedule

Specifies the type of scheduling algorithms and, except in the case of **auto**, chunk size (n) that are used for loops to which no other scheduling algorithm has been explicitly assigned in the source code. Suboptions of the **schedule** suboption are as follows:

affinity[=n]

The iterations of a loop are initially divided into n partitions, containing **ceiling**(*number_of_iterations*/*number_of_threads*) iterations. Each partition is initially assigned to a thread and is then further subdivided into chunks that each contain n iterations. If n is not specified, then the chunks consist of **ceiling**(*number_of_iterations_left_in_partition* / 2) loop iterations.

When a thread becomes free, it takes the next chunk from its initially assigned partition. If there are no more chunks in that partition, then the thread takes the next available chunk from a partition initially assigned to another thread.

The work in a partition initially assigned to a sleeping thread will be completed by threads that are active.

The **affinity** scheduling type does not appear in the OpenMP API standard.

auto
Scheduling of the loop iterations is delegated to the compiler and runtime systems. The compiler and runtime system can choose any possible mapping of iterations to threads (including all possible valid schedule types) and these might be different in different loops. Do not specify chunk size (*n*).

dynamic[=*n*]
The iterations of a loop are divided into chunks containing *n* iterations each. If *n* is not specified, then the chunks consist of **ceiling**(*number_of_iterations*/*number_of_threads*). iterations.

Active threads are assigned these chunks on a "first-come, first-do" basis. Chunks of the remaining work are assigned to available threads until all work has been assigned.

If a thread is asleep, its assigned work will be taken over by an active thread once that thread becomes available.

guided[=*n*]
The iterations of a loop are divided into progressively smaller chunks until a minimum chunk size of *n* loop iterations is reached. If *n* is not specified, the default value for *n* is 1 iteration.

Active threads are assigned chunks on a "first-come, first-do" basis. The first chunk contains **ceiling**(*number_of_iterations*/*number_of_threads*) iterations. Subsequent chunks consist of **ceiling**(*number_of_iterations_left* / *number_of_threads*) iterations.

runtime
Specifies that the chunking algorithm will be determined at run time.

static[=*n*]
The iterations of a loop are divided into chunks containing *n* iterations each. Each thread is assigned chunks in a "round-robin" fashion. This is known as *block cyclic scheduling*. If the value of *n* is 1, then the scheduling type is specifically referred to as *cyclic scheduling*.

If *n* is not specified, the chunks will contain **ceiling**(*number_of_iterations*/ *number_of_threads*) iterations. Each thread is assigned one of these chunks. This is known as *block scheduling*.

If a thread is asleep and it has been assigned work, it will be awakened so that it may complete its work.

n   Must be an integer of value 1 or greater.

Specifying **schedule** with no suboption is equivalent to **schedule=runtime**.

stackcheck | nostackcheck
Causes the compiler to check for stack overflow by slave threads at run time, and issue a warning if the remaining stack size is less than the number of bytes specified by the **stackcheck** option of the XLSMPOPTS environment variable. This suboption is intended for debugging purposes, and only takes effect when **XLSMPOPTS=stackcheck** is also set; see "XLSMPOPTS" on page 25.

threshold[=*n*]
When **-qsmp=auto** is in effect, controls the amount of automatic loop

parallelization that occurs. The value of n represents the minimum amount of work required in a loop in order for it to be parallelized. Currently, the calculation of "work" is weighted heavily by the number of iterations in the loop. In general, the higher the value specified for n, the fewer loops are parallelized. Specifying a value of 0 instructs the compiler to parallelize all auto-parallelizable loops, whether or not it is profitable to do so. Specifying a value of 100 instructs the compiler to parallelize only those auto-parallelizable loops that it deems profitable. Specifying a value of greater than 100 will result in more loops being serialized.

n Must be a positive integer of 0 or greater.

If you specify **threshold** with no suboption, the program uses a default value of 100.

Specifying **-qsmp** without suboptions is equivalent to:
```
-qsmp=auto:explicit:opt:noomp:norec_locks:nonested_par:schedule=runtime:
nostackcheck:threshold=100:ostls
```

Usage
- Specifying the **omp** suboption always implies **noauto**. Specify **-qsmp=omp:auto** to apply automatic parallelization on OpenMP-compliant applications, as well.
- You should only use **-qsmp** with the **_r**-suffixed invocation commands, to automatically link in all of the threadsafe components. You can use the **-qsmp** option with the non-**_r**-suffixed invocation commands, but you are responsible for linking in the appropriate components. . If you use the **-qsmp** option to compile any source file in a program, then you must specify the **-qsmp** option at link time as well, unless you link by using the **ld** command.
- Object files generated with the **-qsmp=opt** option can be linked with object files generated with **-qsmp=noopt**. The visibility within the debugger of the variables in each object file will not be affected by linking.
- The **-qnosmp** default option setting specifies that no code should be generated for parallelization directives, though syntax checking will still be performed. Use **-qignprag=omp** to completely ignore parallelization directives.
- Specifying **-qsmp** implicitly sets **-O2**. The **-qsmp** option overrides **-qnooptimize**, but does not override **-O3**, **-O4**, or **-O5**. When debugging parallelized program code, you can disable optimization in parallelized program code by specifying **qsmp=noopt**.
- The **-qsmp=noopt** suboption overrides performance optimization options anywhere on the command line unless **-qsmp** appears after **-qsmp=noopt**. For example, **-qsmp=noopt -O3** is equivalent to **-qsmp=noopt**, while **-qsmp=noopt -O3 -qsmp** is equivalent to **-qsmp -O3**.

Related information
- "-O, -qoptimize" on page 227
- "-qthreaded" on page 302
- "Environment variables for parallel processing" on page 25
- "Pragma directives for parallel processing" on page 377
- "Built-in functions for parallel processing" on page 506

-qsource
Category

Listings, messages, and compiler information

Pragma equivalent

#pragma options [no]source

Purpose

Produces a compiler listing file that includes the source section of the listing and provides additional source information when printing error messages.

When **source** is in effect, a listing file is generated with a .lst suffix for each source file named on the command line. For details of the contents of the listing file, see "Compiler listings" on page 19.

Syntax

```
►►── -q──┬─nosource─┬──────────────────────────────────────────────►◄
         └─source───┘
```

Defaults

-qnosource

Usage

You can selectively print parts of the source by using pairs of **#pragma options source** and **#pragma options nosource** preprocessor directives throughout your source program. The source following **#pragma options source** and preceding **#pragma options nosource** is printed.

The **-qnoprint** option overrides this option.

Predefined macros

None.

Examples

To compile myprogram.c to produce a compiler listing that includes the source code, enter:

```
xlc myprogram.c -qsource
```

Related information
- "-qlist" on page 210
- "-qlistopt" on page 214
- "-qprint" on page 248

-qsourcetype
Category

Input control

Pragma equivalent

None.

Purpose

Instructs the compiler to treat all recognized source files as a specified source type, regardless of the actual file name suffix.

Ordinarily, the compiler uses the file name suffix of source files specified on the command line to determine the type of the source file. For example, a .c suffix normally implies C source code, and a .C suffix normally implies C++ source code. The **-qsourcetype** option instructs the compiler to not rely on the file name suffix, and to instead assume a source type as specified by the option.

Syntax

```
                              ┌─default─────────┐
►►── -q─sourcetype──=──┼─assembler───────┼──────────────────────────────────►◄
                              ├─assembler-with-cpp─┤
                              ├─c───────────────┤
                              └─c++─────────────┘
```
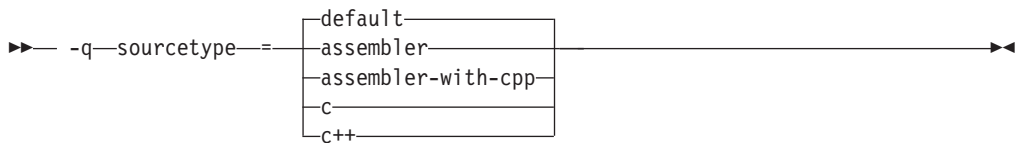
Defaults

-qsourcetype=default

Parameters

assembler
  All source files following the option are compiled as if they are assembler language source files.

assembler-with-cpp
  All source files following the option are compiled as if they are assembler language source files that need preprocessing.

c All source files following the option are compiled as if they are C language source files.

> C++ **c++**
  All source files following the option are compiled as if they are C++ language source files. This suboption is equivalent to the **-+** option.

default
  The programming language of a source file is implied by its file name suffix.

Usage

If you do not use this option, files must have a suffix of .c to be compiled as C files, and .C (uppercase C), .cc, .cp, .cpp, .cxx, or .c++ to be compiled as C++ files.

This option applies whether the file system is case-sensitive or not. That is, even in a case-insensitive file system, where `file.c` and `file.C` refer to the same physical file, the compiler still recognizes the case difference of the file name argument on the command line and determines the source type accordingly.

Note that the option only affects files that are specified on the command line *following* the option, but not those that precede the option. Therefore, in the following example:

```
xlc goodbye.C -qsourcetype=c hello.C
```

`hello.C` is compiled as a C source file, but `goodbye.C` is compiled as a C++ file.

The **-qsourcetype** option should not be used together with the **-+** option.

Predefined macros

None.

Examples

To treat the source file `hello.C` as being a C language source file, enter:
```
xlc -qsourcetype=c hello.C
```

Related information
- "-+ (plus sign) (C++ only)" on page 85

-qspill

Category

Compiler customization

Pragma equivalent

#pragma options [no]spill

Purpose

Specifies the size (in bytes) of the register spill space, the internal program storage areas used by the optimizer for register spills to storage.

Syntax

▶▶── -q─spill──=─*size*──▶◀

Defaults

-qspill=512

Parameters

size
    An integer representing the number of bytes for the register allocation spill area.

Usage

If your program is very complex, or if there are too many computations to hold in registers at one time and your program needs temporary storage, you might need to increase this area. Do not enlarge the spill area unless the compiler issues a message requesting a larger spill area. In case of a conflict, the largest spill area specified is used.

Predefined macros

None.

Examples

If you received a warning message when compiling `myprogram.c` and want to compile it specifying a spill area of 900 entries, enter:

```
xlc myprogram.c -qspill=900
```

-qsrcmsg (C only)
Category

Listings, messages, and compiler information

Pragma equivalent

#pragma options [no]srcmsg

Purpose

Adds the corresponding source code lines to diagnostic messages generated by the compiler.

When **nosrcmsg** is in effect, the error message simply shows the file, line and column where the error occurred. When **srcmsg** is in effect, the compiler reconstructs the source line or partial source line to which the diagnostic message refers and displays it before the diagnostic message. A pointer to the column position of the error may also be displayed.

Syntax

```
                 ┌─nosrcmsg─┐
►►── -q──────────┴─srcmsg───┴─────────────────────────────────────────►◄
```

Defaults

-qnosrcmsg

Usage

When **srcmsg** is in effect, the reconstructed source line represents the line as it appears after macro expansion. At times, the line may be only partially reconstructed. The characters "`....`" at the start or end of the displayed line indicate that some of the source line has not been displayed.

Use **-qnosrcmsg** to display concise messages that can be parsed.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the source line is displayed along with the diagnostic message when an error occurs, enter:

```
xlc myprogram.c -qsrcmsg
```

-qstackprotect

Category

"Object code control" on page 72

Pragma equivalent

None.

Purpose

Provides protection against malicious code or programming errors that overwrite or corrupt the stack.

Syntax

```
          ┌─nostackprotect─┐
►►── -q────┴─stackprotect──┴────┬─all──────────┬──────────────────►◄
                            =   └─size──=──N────┘
```
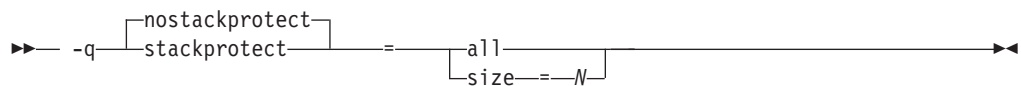
Defaults

- -qnostackprotect

Parameters

all

    all protects all procedures whether or not there are vulnerable objects. This option is not set by default.

size=_N_

    size=_N_ protects all procedures containing automatic objects greater or equal to _N_ bytes in size. The default size is 8 when **-qstackprotect** is enabled.

    Note: When both **all** and **size** are used, the last option wins.

Usage

-qstackprotect generates extra code to protect procedures with vulnerable objects against stack corruption. This option is disabled by default because it can cause performance degradation. The default option is **-qnostackprotect**.

To generate code to protect all procedures with vulnerable objects:

```
xlc myprogram.c -qstackprotect=all
```

To generate code to protect procedures with objects of certain bytes:

```
xlc myprogram.c -qstackprotect=size=8
```

Note:
- This option cannot be used with #pragma options.
- Because of the dependency on **glibc** in Linux, this option requires the following Linux levels:
  - Linux OS with **GLIBC** 2.4 and up (Machines with **GCC** 4.X and up).
- All supported Linux systems already support this feature.

Predefined macros

None.

Related information

- "-qinfo" on page 164

-qstaticinline (C++ only)
Category

Language element control

Pragma equivalent

None.

Purpose

Controls whether inline functions are treated as having `static` or `extern` linkage.

When **-qnostaticinline** is in effect, the compiler treats inline functions as `extern`: only one function body is generated for a function marked with the `inline` function specifier, regardless of how many definitions of the same function appear in different source files. When **-qstaticinline** is in effect, the compiler treats inline functions as having `static` linkage: a separate function body is generated for each definition in a different source file of the same function marked with the `inline` function specifier.

Syntax

```
               ┌─nostaticinline─┐
►►── -q────────┼─staticinline───┤──────────────────────────────────►◄
```

Defaults

-qnostaticinline

Usage

When **-qnostaticinline** is in effect, any redundant functions definitions for which no bodies are generated are discarded by default; you can use the **-qkeepinlines** option to change this behavior.

Predefined macros

None.

Examples

Using the **-qstaticinline** option causes function f in the following declaration to be treated as static, even though it is not explicitly declared as such. A separate function body is created for each definition of the function. Note that this can lead to a substantial increase in code size.

```
inline void f() {/*...*/};
```

Related information

- "Linkage of inline functions" in the *XL C/C++ Language Reference*

-qstaticlink

Category

Linking

Pragma equivalent

None.

Purpose

Controls how shared and nonshared runtime libraries are linked into an application.

When **-qstaticlink** is in effect, the compiler links only static libraries with the object file being produced. When **-qnostaticlink** is in effect, the compiler links shared libraries with the object file being produced.

This option provides the ability to specify linking rules that are equivalent to those implied by the GNU options **-static**, **-static-libgcc**, and **-shared-libgcc**, used singly and in combination.

Syntax

```
            ┌─nostaticlink─┐
►►── -q──┴─staticlink───┴──────────────────────────────────►◄
                          └─=──libgcc─┘
```

Defaults

-qnostaticlink

Parameters

libgcc
>   When **libgcc** is specified together with **nostaticlink**, the compiler links to the shared version of **libgcc**. When **libgcc** is specified together with **staticlink**, the compiler links to the static version of **libgcc**.

Usage

Important: Any use of third-party libraries or products is subject to the provisions in their respective licenses. Using the **-qstaticlink** option can have significant legal consequences for the programs that you compile. It is strongly recommended that you seek legal advice before using this option.

The following table shows the equivalent GNU and XL C/C++XL Fortran options for specifying linkage of shared and nonshared libraries.

Table 27. Option mappings: control of the GNU linker

GNU option	Meaning	XL Fortran option
-shared	Build a shared object.	-qmkshrobj
-static	Build a static object and prevent linking with shared libraries. Every library linked to must be a static library.	-qstaticlink
-shared-libgcc	Link with the shared version of libgcc.	-qnostaticlink or -qnostaticlink=libgcc (these two are identical) **Note:** This is the default setting on SUSE Linux Enterprise Server (SLES) and Red Hat Enterprise Linux (RHEL).
-static-libgcc	Link with the static version of libgcc. You can still link your shared libraries.	-qstaticlink=libgcc

Note: Options **-qmkshrobj** and **-qstaticlink** are incompatible and cannot be specified together.

Predefined macros

None.

Related information
- "-qmkshrobj" on page 224

-qstatsym
Category

Object code control

Pragma equivalent

None.

Purpose

Adds user-defined, nonexternal names that have a persistent storage class, such as initialized and uninitialized static variables, to the symbol table of the object file.

Syntax

```
           ┌─nostatsym─┐
►►── -q────┴─statsym───┴──────────────────────────────────────►◄
```

Defaults

-qnostatsym: Static variables are not added to the symbol table. However, static functions are added to the symbol table.

Predefined macros

None.

Examples

To compile `myprogram.c` so that static symbols are added to the symbol table, enter:

```
xlc myprogram.c -qstatsym
```

-qstdinc

Category

Input control

Pragma equivalent

#pragma options [no]stdinc

Purpose

Specifies whether the standard include directories are included in the search paths for system and user header files.

When **-qstdinc** is in effect, the compiler searches the following directories for header files:

- ▶ **C** The directory specified in the configuration file for the XL C header files (this is normally /opt/ibmcmp/vacpp/11.1/include/) or by the **-qc_stdinc** option
- ▶ **C++** The directory specified in the configuration file for the XL C and C++ header files (this is normally /opt/ibmcmp/vacpp/11.1/include/) or by the **-qcpp_stdinc** option
- The directory specified in the configuration file for the system header files or by the **-qgcc_c_stdinc** and **-qgcc_cpp_stdinc** options

When **-qnostdinc** is in effect, these directories are excluded from the search paths. The only directories to be searched are:

- directories in which source files containing `#include "filename"` directives are located
- directories specified by the **-I** option
- directories specified by the **-qinclude** option

Syntax

```
         ┌─stdinc───┐
►►── -q───┴─nostdinc─┴──────────────────────────────────►◄
```

Defaults

-qstdinc

Usage

The search order of header files is described in "Directory search sequence for include files" on page 12.

This option only affects search paths for header files included with a relative name; if a full (absolute) path name is specified, this option has no effect on that path name.

The last valid pragma directive remains in effect until replaced by a subsequent pragma.

Predefined macros

None.

Examples

To compile myprogram.c so that *only* the directory /tmp/myfiles (in addition to the directory containing myprogram.c) is searched for the file included with the #include "myinc.h" directive, enter:

```
xlc myprogram.c -qnostdinc -I/tmp/myfiles
```

Related information
- "-qc_stdinc (C only)" on page 118
- "-qcpp_stdinc (C++ only)" on page 119
- "-qgcc_c_stdinc (C only)" on page 150
- "-qgcc_cpp_stdinc (C++ only)" on page 151
- "-I" on page 158
- "Directory search sequence for include files" on page 12

-qstrict

Category

Optimization and tuning

Pragma equivalent

#pragma options [no]strict

#pragma option_override (*function_name*, "opt (*suboption_list*)")

Purpose

Ensures that optimizations done by default at optimization levels **-O3** and higher, and, optionally at **-O2**, do not alter the semantics of a program.

This option is intended for situations where the changes in program execution in optimized programs produce different results from unoptimized programs.

Syntax

```
►►── -q──┬─nostrict──────────────────────────────────────────────────────────────┬──►◄
         └─strict──┬──────────────────────────────────────────────────────────┐
                   │                         ┌─ : ──────────────────┐
                   │                         │  ▼                   │
                   └─ = ──┬──────┬─────────────all────────────────┬──
                                 ├─none────────────────┤
                                 ├─precision───────────┤
                                 ├─noprecision─────────┤
                                 ├─exceptions──────────┤
                                 ├─noexceptions────────┤
                                 ├─ieeefp──────────────┤
                                 ├─noieeefp────────────┤
                                 ├─nans────────────────┤
                                 ├─nonans──────────────┤
                                 ├─infinities──────────┤
                                 ├─noinfinities────────┤
                                 ├─subnormals──────────┤
                                 ├─nosubnormals────────┤
                                 ├─zerosigns───────────┤
                                 ├─nozerosigns─────────┤
                                 ├─operationprecision──┤
                                 ├─nooperationprecision┤
                                 ├─vectorprecision─────┤
                                 ├─novectorprecision───┤
                                 ├─order───────────────┤
                                 ├─noorder─────────────┤
                                 ├─association─────────┤
                                 ├─noassociation───────┤
                                 ├─reductionorder──────┤
                                 ├─noreductionorder────┤
                                 ├─guards──────────────┤
                                 ├─noguards────────────┤
                                 ├─library─────────────┤
                                 └─nolibrary───────────┘
```

Defaults

- Always **-qstrict** or **-qstrict=all** when the **-qnoopt** or **-O0** optimization level is in effect
- **-qstrict** or **-qstrict=all** is the default when the **-O2** or **-O** optimization level is in effect
- **-qnostrict** or **-qstrict=none** is the default when **-O3** or a higher optimization level is in effect

Parameters

The **-qstrict** suboptions include the following:

all | none
    all disables all semantics-changing transformations, including those controlled by the **ieeefp**, **order**, **library**, **precision**, and **exceptions** suboptions. **none** enables these transformations.

precision | noprecision
    precision disables all transformations that are likely to affect floating-point precision, including those controlled by the **subnormals**, **operationprecision**, **vectorprecision**, **association**, **reductionorder**, and **library** suboptions. **noprecision** enables these transformations.

exceptions | noexceptions

exceptions disables all transformations likely to affect exceptions or be affected by them, including those controlled by the **nans**, **infinities**, **subnormals**, **guards**, and **library** suboptions. **noexceptions** enables these transformations.

ieeefp | noieeefp

ieeefp disables transformations that affect IEEE floating-point compliance, including those controlled by the **nans**, **infinities**, **subnormals**, **zerosigns**, **vectorprecision**, and **operationprecision** suboptions. **noieeefp** enables these transformations.

nans | nonans

nans disables transformations that may produce incorrect results in the presence of, or that may incorrectly produce IEEE floating-point NaN (not-a-number) values. **nonans** enables these transformations.

infinities | noinfinities

infinities disables transformations that may produce incorrect results in the presence of, or that may incorrectly produce floating-point infinities. **noinfinities** enables these transformations.

subnormals | nosubnormals

subnormals disables transformations that may produce incorrect results in the presence of, or that may incorrectly produce IEEE floating-point subnormals (formerly known as denorms). **nosubnormals** enables these transformations.

zerosigns | nozerosigns

zerosigns disables transformations that may affect or be affected by whether the sign of a floating-point zero is correct. **nozerosigns** enables these transformations.

operationprecision | nooperationprecision

operationprecision disables transformations that produce approximate results for individual floating-point operations. **nooperationprecision** enables these transformations.

vectorprecision | novectorprecision

vectorprecision disables vectorization in loops where it might produce different results in vectorized iterations than in nonvectorized residue iterations. **vectorprecision** ensures that every loop iteration of identical floating point operations on identical data produces identical results.

novectorprecision enables vectorization even when different iterations might produce different results from the same inputs.

order | noorder

order disables all code reordering between multiple operations that may affect results or exceptions, including those controlled by the **association**, **reductionorder**, and **guards** suboptions. **noorder** enables code reordering.

association | noassociation

association disables reordering operations within an expression. **noassociation** enables reordering operations.

reductionorder | noreductionorder

reductionorder disables parallelizing floating-point reductions. **noreductionorder** enables parallelizing these reductions.

guards | noguards

guards disables moving operations past guards (that is, past **if**, out of loops, or

past function calls that might end the program or throw an exception) which control whether the operation should be executed. **noguards** enables moving operations past guards.

library | nolibrary

    library disables transformations that affect floating-point library functions; for example, transformations that replace floating-point library functions with other library functions or with constants. **nolibrary** enables these transformations.

Usage

The **all**, **precision**, **exceptions**, **ieeefp**, and **order** suboptions and their negative forms are group suboptions that affect multiple, individual suboptions. For many situations, the group suboptions will give sufficient granular control over transformations. Group suboptions act as if either the positive or the no form of every suboption of the group is specified. Where necessary, individual suboptions within a group (like **subnormals** or **operationprecision** within the **precision** group) provide control of specific transformations within that group.

With **-qnostrict** or **-qstrict=none** in effect, the following optimizations are turned on:

- Code that may cause an exception may be rearranged. The corresponding exception might happen at a different point in execution or might not occur at all. (The compiler still tries to minimize such situations.)
- Floating-point operations may not preserve the sign of a zero value. (To make certain that this sign is preserved, you also need to specify **-qfloat=rrm**, **-qfloat=nomaf**, or **-qfloat=strictnmaf**.)
- Floating-point expressions may be reassociated. For example, **(2.0*3.1)*4.2** might become **2.0*(3.1*4.2)** if that is faster, even though the result might not be identical.
- The **fltint** and **rsqrt** suboptions of the **-qfloat** option are turned on. You can turn them off again by also using the **-qstrict** option or the **nofltint** and **norsqrt** suboptions of **-qfloat**. With lower-level or no optimization specified, these suboptions are turned off by default.

Specifying various **-qstrict[=suboptions]** or **-qnostrict** combinations sets the following suboptions:

- **-qstrict** or **-qstrict=all** sets **-qfloat=norsqrt:rngchk**. **-qnostrict** or **-qstrict=none** sets **-qfloat=rsqrt:norngchk**.
- **-qstrict=operationprecision** or **-qstrict=exceptions** sets **-qfloat=nofltint**. Specifying both **-qstrict=nooperationprecision** and **-qstrict=noexceptions** sets **-qfloat=fltint**.
- **-qstrict=infinities**, **-qstrict=operationprecision**, or **-qstrict=exceptions** sets **-qfloat=norsqrt**.
- **-qstrict=noinfinities:nooperationprecision:noexceptions** sets **-qfloat=rsqrt**.
- **-qstrict=nans**, **-qstrict=infinities**, **-qstrict=zerosigns**, or **-qstrict=exceptions** sets **-qfloat=rngchk**. Specifying all of **-qstrict=nonans:nozerosigns:noexceptions** or **-qstrict=noinfinities:nozerosigns:noexceptions**, or any group suboptions that imply all of them, sets **-qfloat=norngchk**.

Note: For details about the relationship between **-qstrict** suboptions and their **-qfloat** counterparts, see "-qfloat" on page 137.

To override any of these settings, specify the appropriate **-qfloat** suboptions after the **-qstrict** option on the command line.

Predefined macros

None.

Examples

To compile myprogram.c so that the aggressive optimizations of **-O3** are turned off, and division by the result of a square root is replaced by multiplying by the reciprocal (**-qfloat=rsqrt**), enter:

```
xlc myprogram.c -O3 -qstrict -qfloat=rsqrt
```

To enable all transformations except those affecting precision, specify:

```
xlc myprogram.c -qstrict=none:precision
```

To disable all transformations except those involving NaNs and infinities, specify:

```
xlc myprogram.c -qstrict=all:nonans:noinfinities
```

Related information
- "-qsimd" on page 269
- "-qfloat" on page 137
- "-qhot" on page 155
- "-O, -qoptimize" on page 227

-qstrict_induction
Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Prevents the compiler from performing induction (loop counter) variable optimizations. These optimizations may be unsafe (may alter the semantics of your program) when there are integer overflow operations involving the induction variables.

Syntax

```
        ┌─strict_induction──┐
►►─ -q───┴─nostrict_induction─┴──────────────────────────────►◄
```

Defaults
- **-qstrict_induction**
- **-qnostrict_induction** when **-O2** or higher optimization level is in effect

Usage

When using **-O2** or higher optimization, you can specify **-qstrict_induction** to prevent optimizations that change the result of a program if truncation or sign extension of a loop induction variable should occur as a result of variable overflow or wrap-around. However, use of **-qstrict_induction** is generally not recommended because it can cause considerable performance degradation.

Predefined macros

None.

Related information
- "-O, -qoptimize" on page 227

-qsuppress

Category

Listings, messages, and compiler information

Pragma equivalent

"#pragma report (C++ only)" on page 368

Purpose

Prevents specific informational or warning messages from being displayed or added to the listing file, if one is generated.

Syntax

```
                   ┌─nosuppress─┐
                   │            │
                   │      ┌─:─────────────────┐
                   │      ▼                   │
►►── -q ──┬────────┴───────────────────────────┬──►◄
          └─ suppress ── = ──┬─ message_identifier ─┴─
```
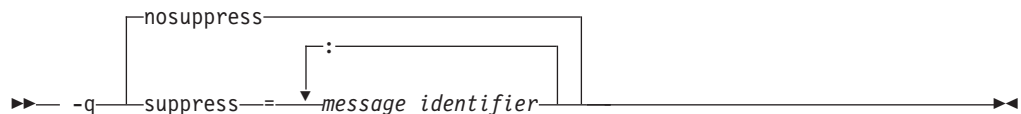
Defaults

-qnosuppress: All informational and warning messages are reported, unless set otherwise with the **-qflag** option.

Parameters

message_identifier
Represents a message identifier. The message identifier must be in the following format:

15*dd-number*

where:

dd   Is the two-digit code representing the compiler component that produces the message. See "Compiler message format" on page 16 for descriptions of these.

number
Is the message number.

Usage

You can only suppress information (I) and warning (W) messages. You cannot suppress other types of messages, such as (S) and (U) level messages. Note that informational and warning messages that supply additional information to a severe error cannot be disabled by this option.

To suppress all informational and warning messages, you can use the **-w** option.

To suppress IPA messages, enter **-qsuppress** before **-qipa** on the command line.

The **-qnosuppress** compiler option cancels previous settings of **-qsuppress**.

Predefined macros

None.

Examples

If your program normally results in the following output:

```
"myprogram.c", line 1.1:1506-224 (I) Incorrect #pragma ignored
```

you can suppress the message by compiling with:

```
xlc myprogram.c -qsuppress=1506-224
```

Related information
- "-qflag" on page 136

-qsymtab (C only)
Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Determines the information that appears in the symbol table.

Syntax

```
►►── -q─symtab──=──┬─unref─┬────────────────────────────────────────►◄
                   └─static─┘
```

Defaults

Static variables and unreferenced `typedef`, structure, union, and enumeration declarations are not included in the symbol table of the object file.

Parameters

unref
> When used with the **-g** option, specifies that debugging information is

included for unreferenced `typedef` declarations, `struct`, `union`, and `enum` type definitions in the symbol table of the object file. This suboption is equivalent to **-qdbxextra**.

Using **-qsymtab=unref** may make your object and executable files larger.

static
Adds user-defined, nonexternal names that have a persistent storage class, such as initialized and uninitialized static variables, to the symbol table of the object file. This suboption is equivalent to **-qstatsym**.

Predefined macros

None.

Examples

To compile `myprogram.c` so that static symbols are added to the symbol table, enter:

```
xlc myprogram.c -qsymtab=static
```

To compile `myprogram.c` so that unreferenced `typedef`, structure, union, and enumeration declarations are included in the symbol table for use with a debugger, enter:

```
xlc myprogram.c -g -qsymtab=unref
```

Related information
- "-g" on page 149
- "-qdbxextra (C only)" on page 123
- "-qstatsym" on page 284

-qsyntaxonly (C only)
Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Performs syntax checking without generating an object file.

Syntax

▶▶── -q─syntaxonly───▶◀

Defaults

By default, source files are compiled and linked to generate an executable file.

Usage

The **-P**, **-E**, and **-C** options override the **-qsyntaxonly** option, which in turn overrides the **-c** and **-o** options.

The **-qsyntaxonly** option suppresses only the generation of an object file. All other files, such as listing files, are still produced if their corresponding options are set.

Predefined macros

None.

Examples

To check the syntax of `myprogram.c` without generating an object file, enter:

```
xlc myprogram.c -qsyntaxonly
```

Related information
- "-C, -C!" on page 106
- "-c" on page 105
- "-E" on page 127
- "-o" on page 226
- "-P" on page 232

-t

Category

Compiler customization

Pragma equivalent

None.

Purpose

Applies the prefix specified by the **-B** option to the designated components.

Syntax

Defaults

The default paths for all of the compiler executables are defined in the compiler configuration file.

Parameters

The following table shows the correspondence between **-t** parameters and the component executable names:

Parameter	Description	Executable name
a	Assembler	as
b	Low-level optimizer	xlCcode
c	Compiler front end	xlcentry, xlCentry
▶ C++ C	C++ compiler front end	xlCentry
d	Disassembler	dis
I	High-level optimizer, compile step	ipa
L	High-level optimizer, link step	ipa
l	Linker	ld

Usage

This option is intended to be used together with the **-B***prefix* option. If **-B** is specified without the *prefix*, the default prefix is /lib/o. If **-B** is not specified at all, the prefix of the standard program names is /lib/n.

Note: If you use the **p** suboption, it can cause the source code to be preprocessed separately before compilation, which can change the way a program is compiled.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the name `/u/newones/compilers/` is prefixed to the compiler and assembler program names, enter:

```
xlc myprogram.c -B/u/newones/compilers/ -tca
```

Related information
* "-B" on page 103

-qtabsize

Category

Language element control

Pragma equivalent

#pragma options tabsize

Purpose

Sets the default tab length, for the purposes of reporting the column number in error messages.

Syntax

```
►►── -q─tabsize──=───number─────────────────────────────────────────────────►◄
```

Defaults

-qtabsize=8

Parameters

number
> The number of character spaces representing a tab in your source program.

Usage

This option only affects error messages that specify the column number at which an error occurred.

Predefined macros

None.

Examples

To compile myprogram.c so the compiler considers tabs as having a width of one character, enter:

```
xlc myprogram.c -qtabsize=1
```

In this case, you can consider one character position (where each character and each tab equals one position, regardless of tab length) as being equivalent to one character column.

-qtbtable
Category

Object code control

Pragma equivalent

#pragma options tbtable

Purpose

Controls the amount of debugging traceback information that is included in the object files.

Many performance measurement tools require a full traceback table to properly analyze optimized code. If a traceback table is generated, it is placed in the text segment at the end of the object code, and contains information about each function, including the type of function, as well as stack frame and register information.

Syntax

```
►►── -q─tbtable──=──┬─full──┬───────────────────────────────────────────────◄◄
                    ├─none──┤
                    └─small─┘
```

Defaults

- **-qtbtable=full**
- **-qtbtable=small** when **-O** or higher optimization is in effect

Parameters

full

A full traceback table is generated, complete with name and parameter information.

none

No traceback table is generated. The stack frame cannot be unwound so exception handling is disabled.

small

The traceback table generated has no name or parameter information, but otherwise has full traceback capability. This suboption reduces the size of the program code.

Usage

This option applies only to 64-bit compilations, and is ignored if specified for a 32-bit compilation.

The **#pragma** options directive must be specified before the first statement in the compilation unit.

Predefined macros

None.

Related information
-

-qtempinc (C++ only)
Category

Template control

Pragma equivalent

None.

Purpose

Generates separate template include files for template functions and class declarations, and places these files in a directory which can be optionally specified.

Syntax

```
►►── -q ──┬─notempinc─────────────────────────────┬──────────────────►◄
          ├─tempinc──────────────────────────────┤
                   └─=──directory_path─┘
```
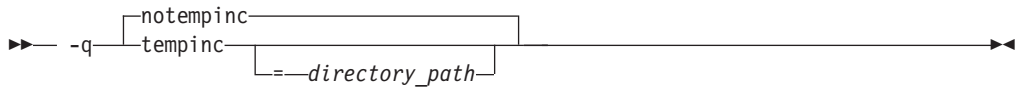
Defaults

-qnotempinc

Parameters

directory_path
>   The directory in which the generated template include files are to be placed.

Usage

The **-qtempinc** and **-qtemplateregistry** compiler options are mutually exclusive. Specifying **-qtempinc** implies **-qnotemplateregistry**. Similarly, specifying **-qtemplateregistry** implies **-qnotempinc**. However, specifying **-qnotempinc** does not imply **-qtemplateregistry**.

Specifying either **-qtempinc** or **-qtemplateregistry** implies **-qtmplinst=auto**.

Predefined macros

__TEMPINC__ is predefined to 1 when **-qtempinc** is in effect; otherwise, it is not defined.

Examples

To compile the file myprogram.C and place the generated include files for the template functions in the /tmp/mytemplates directory, enter:

```
 xlc++ myprogram.C -qtempinc=/tmp/mytemplates
```

Related information
- "#pragma implementation (C++ only)" on page 348
- "-qtmplinst (C++ only)" on page 305
- "-qtemplateregistry (C++ only)" on page 300
- "-qtemplaterecompile (C++ only)" on page 299
- "Using C++ templates" in the *XL C/C++ Optimization and Programming Guide*.

-qtemplatedepth (C++ only)
Category

Template control

Pragma equivalent

None.

Purpose

Specifies the maximum number of recursively instantiated template specializations that will be processed by the compiler.

Syntax

```
►►── -q─templatedepth──=─number──────────────────────────────────────────◄◄
```

Defaults

-qtemplatedepth=300

Parameters

number
>   The maximum number of recursive template instantiations. The number can be
>   a value between 1 and INT_MAX. If your code attempts to recursively
>   instantiate more templates than *number*, compilation halts and an error
>   message is issued. If you specify an invalid value, the default value of 300 is
>   used.

Usage

Note that setting this option to a high value can potentially cause an
out-of-memory error due to the complexity and amount of code generated.

Predefined macros

None.

Examples

To allow the following code in `myprogram.cpp` to be compiled successfully:

```
template <int n> void foo() {
   foo<n-1>();
}

template <> void foo<0>() {}

   int main() {
   foo<400>();
}
```

Enter:

```
xlc++ myprogram.cpp -qtemplatedepth=400
```

Related information
- "Using C++ templates" in the *XL C/C++ Optimization and Programming Guide*.

-qtemplaterecompile (C++ only)
Category

Template control

Pragma equivalent

None.

Purpose

Helps manage dependencies between compilation units that have been compiled using the **-qtemplateregistry** compiler option.

Syntax

```
►►── -q──┬─templaterecompile─┬──────────────────────────────── ►◄
         └─notemplaterecompile─┘
```

Defaults

-qtemplaterecompile

Usage

If a source file that has been compiled previously is compiled again, the **-qtemplaterecompile** option consults the template registry to determine whether changes to this source file require the recompile of other compilation units. This can occur when the source file has changed in such a way that it no longer references a given instantiation and the corresponding object file previously contained the instantiation. If so, affected compilation units will be recompiled automatically.

The **-qtemplaterecompile** option requires that object files generated by the compiler remain in the subdirectory to which they were originally written. If your automated build process moves object files from their original subdirectory, use the **-qnotemplaterecompile** option whenever **-qtemplateregistry** is enabled.

Predefined macros

None.

Related information
- "-qtmplinst (C++ only)" on page 305
- "-qtempinc (C++ only)" on page 297
- "-qtemplateregistry (C++ only)"
- "Using C++ templates" in the *XL C/C++ Optimization and Programming Guide*.

-qtemplateregistry (C++ only)
Category

Template control

Pragma equivalent

None.

Purpose

Maintains records of all templates as they are encountered in the source and ensures that only one instantiation of each template is made.

The first time that the compiler encounters a reference to a template instantiation, that instantiation is generated and the related object code is placed in the current object file. Any further references to identical instantiations of the same template in different compilation units are recorded but the redundant instantiations are not generated. No special file organization is required to use the **-qtemplateregistry** option.

Syntax
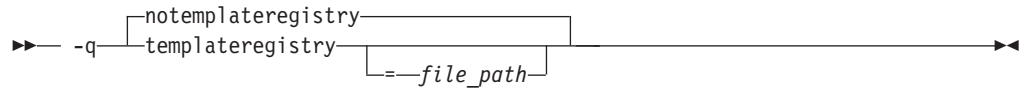
Defaults

-qnotemplateregistry

Parameters

file_path
  The path for the file that will contain the template instantiation information. If you do not specify a location the compiler saves all template registry information to the file `templateregistry` stored in the current working directory.

Usage

Template registry files must not be shared between different programs. If there are two or more programs whose source is in the same directory, relying on the default template registry file stored in the current working directory may lead to incorrect results.

The **-qtempinc** and **-qtemplateregistry** compiler options are mutually exclusive. Specifying **-qtempinc** implies **-qnotemplateregistry**. Similarly, specifying **-qtemplateregistry** implies **-qnotempinc**. However, specifying **-qnotemplateregistry** does not imply **-qtempinc**.

Specifying either **-qtempinc** or **-qtemplateregistry** implies **-qtmplinst=auto**.

Predefined macros

None.

Examples

To compile the file `myprogram.C` and place the template registry information into the `/tmp/mytemplateregistry` file, enter:

```
 xlc++ myprogram.C -qtemplateregistry=/tmp/mytemplateregistry
```

Related information
- "-qtmplinst (C++ only)" on page 305
- "-qtempinc (C++ only)" on page 297
- "-qtemplaterecompile (C++ only)" on page 299
- "Using C++ templates" in the *XL C/C++ Optimization and Programming Guide*.

-qtempmax (C++ only)

Category

Template control

Pragma equivalent

None.

Purpose

Specifies the maximum number of template include files to be generated by the **-qtempinc** option for each header file.

Syntax

►►── -q──tempmax──=──*number*──────────────────────────────────►◄

Defaults

-qtempmax=1

Parameters

number
    The maximum number of template include files. The number can be a value between 1 and 99 999.

Usage

This option should be used when the size of files generated by the **-qtempinc** option become very large and take a significant amount of time to recompile when a new instance is created.

Instantiations are spread among the template include files.

Predefined macros

None.

Related information
- "-qtempinc (C++ only)" on page 297
- "Using C++ templates" in the *XL C/C++ Optimization and Programming Guide*.

-qthreaded

Category

Object code control

Pragma equivalent

None.

Purpose

Indicates to the compiler whether it must generate threadsafe code.

Always use this option when compiling or linking multithreaded applications. This option does not make code threadsafe, but it will ensure that code already threadsafe will remain so after compilation and linking. It also ensures that all optimizations are threadsafe.

Syntax

```
►►── -q──┬─nothreaded─┬──────────────────────────────────────────►◄
         └─threaded───┘
```

Defaults

- **-qnothreaded** for all invocation commands except those with the **_r** suffix
- **-qthreaded** for all **_r**-suffixed invocation commands

Usage

This option applies to both compile and linker operations.

To maintain thread safety, a file compiled with the **-qthreaded** option, whether explicitly by option selection or implicitly by choice of **_r** compiler invocation mode, must also be linked with the **-qthreaded** option.

Predefined macros

None.

Related information

- "-qsmp" on page 272

-qtimestamps

Category

"Output control" on page 67

Pragma equivalent

none.

Purpose

Controls whether or not implicit time stamps are inserted into an object file.

Syntax

```
►►── -q──┬─timestamps───┬──────────────────────────────────────────►◄
         └─notimestamps─┘
```

Defaults

-qtimestamps

Usage

By default, the compiler inserts an implicit time stamp in an object file when it is created. In some cases, comparison tools may not process the information in such binaries properly. Controlling time stamp generation provides a way of avoiding such problems. To omit the time stamp, use the option **-qnotimestamps**.

This option does not affect time stamps inserted by pragmas and other explicit mechanisms.

-qtls

Category

Object code control

Pragma equivalent

None.

Purpose

Enables recognition of the `__thread` storage class specifier, which designates variables that are to be allocated threadlocal storage; and specifies the threadlocal storage model to be used.

When this option is in effect, any variables marked with the `__thread` storage class specifier are treated as local to each thread in a multi-threaded application. At run time, a copy of the variable is created for each thread that accesses it, and destroyed when the thread terminates. Like other high-level constructs that you can use to parallelize your applications, thread-local storage prevents race conditions to global data, without the need for low-level synchronization of threads.

Suboptions allow you to specify thread-local storage models, which provide better performance but are more restrictive in their applicability.

Syntax

```
              ┌─tls─┐
►►── -q──┬─────┴─────┴──────────────────────────────►◄
         │         ┌─default───────┐
         └──=──────┼─global-dynamic─┤
         │         ├─initial-exec──┤
         │         ├─local-exec────┤
         │         ├─local-dynamic─┤
         │         └─unsupported───┘
         └─notls─────────────────────────────────
```
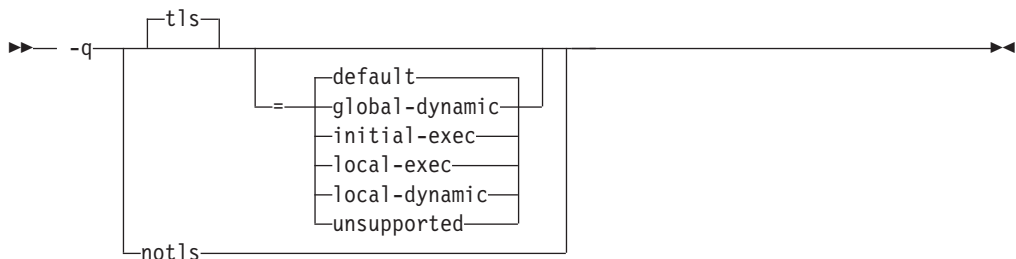
Defaults

-qtls=default

Parameters

unsupported

> The __thread keyword is not recognized and thread-local storage is not enabled. This suboption is equivalent to **-qnotls**.

global-dynamic

> This model is the most general, and can be used for all thread-local variables.

initial-exec

> This model provides better performance than the global-dynamic or local-dynamic models, and can be used for thread-local variables defined in dynamically-loaded modules, provided that those modules are loaded at the same time as the executable. That is, it can only be used when all thread-local variables are defined in modules that are not loaded through dlopen.

local-dynamic

> This model provides better performance than the global-dynamic model, and can be used for thread-local variables defined in dynamically-loaded modules. However, it can only be used when all references to thread-local variables are contained in the same module in which the variables are defined.

local-exec

> This model provides the best performance of all of the models, but can only be used when all thread-local variables are defined and referenced by the main executable.

default

> Uses the appropriate model depending on the setting of the **-qpic** compiler option, which determines whether position-independent code is generated or not. When **-qpic** is in effect, this suboption results in **-qtls=global-dynamic**. When **-qnopic** is in effect, this suboption results in **-qtls=initial-exec** (**-qpic** is in effect by default in 64-bit mode, and cannot be disabled).

Specifying **-qtls** with no suboption is equivalent to **-qtls=default**.

Predefined macros

None.

Related information

- "-qpic" on page 243
- "The __thread storage class specifier" in the *XL C/C++ Language Reference*

-qtmplinst (C++ only)

Category

Template control

Pragma equivalent

None.

Purpose

Manages the implicit instantiation of templates.

Syntax

```
►►── -q──tmplinst──=──┬─auto────┬──────────────────────◄
                      ├─always──┤
                      ├─noinline┤
                      └─none────┘
```

Defaults

-qtmplinst=auto

Parameters

always
>Instructs the compiler to always perform implicit instantiation. If specified, **-qtempinc** and **-qtemplateregistry** compiler options are ignored.

auto
>Manages the implicit instantiations according to the **-qtempinc** and **-qtemplateregistry** options. If both **-qtempinc** and **-qtemplateregistry** are disabled, implicit instantiation will always be performed; otherwise if one of the options is enabled, the compiler manages the implicit instantiation according to that option.

noinline
>Instructs the compiler to not perform any implicit instantiations. If specified, the **-qtempinc** and **-qtemplateregistry** compiler options are ignored.

none
>Instructs the compiler to instantiate only inline functions. No other implicit instantiation is performed. If specified, **-qtempinc** and **-qtemplateregistry** compiler options are ignored.

Usage

You can also use **#pragma do_not_instantiate** to suppress implicit instantiation of selected template classes. See "#pragma do_not_instantiate (C++ only)" on page 342.

Predefined macros

None.

Related information
- "-qtemplateregistry (C++ only)" on page 300
- "-qtempinc (C++ only)" on page 297
- "#pragma do_not_instantiate (C++ only)" on page 342
- "Explicit instantiation" in the *XL C/C++ Language Reference*

-qtmplparse (C++ only)
Category

Template control

Pragma equivalent

None.

Purpose

Controls whether parsing and semantic checking are applied to template definitions.

Syntax

```
►►── -q──tmplparse──=──┬──no──┬────────────────────────────────────►◄
                       ├─error─┤
                       └─warn──┘
```

Defaults

-qtmplparse=no

Parameters

error
Treats problems in template definitions as errors, even if the template is not instantiated.

no Do not parse template definitions. This reduces the number of errors issued in code written for previous versions of VisualAge C++ and predecessor products.

warn
Parses template definitions and issues warning messages for semantic errors.

Usage

This option applies to template definitions, not their instantiations. Regardless of the setting of this option, error messages are produced for problems that appear outside definitions. For example, messages are always produced for errors found during the parsing or semantic checking of constructs such as the following:

- return type of a function template
- parameter list of a function template

Predefined macros

None.

Related information
- "Using C++ templates" in the *XL C/C++ Optimization and Programming Guide*.

-qtocdata
See "-qdataimported, -qdatalocal, -qtocdata" on page 121.

-qtrigraph
Category

Language element control

Pragma equivalent

None.

Purpose

Enables the recognition of trigraph key combinations to represent characters not found on some keyboards.

Syntax

```
>>-- -q---+-trigraph---+---------------------------------------------><
          +-notrigraph-+
```

Defaults

-qtrigraph

Usage

A trigraph is a combination of three-key character combinations that let you produce a character that is not available on all keyboards. For details, see "Trigraph sequences" in the *XL C/C++ Language Reference*.

> **C++** To override the default **-qtrigraph** setting, you must specify **-qnotrigraph** *after* the **-qlanglvl** option on the command line.

Predefined macros

None.

Related information
- "Trigraph sequences" in the *XL C/C++ Language Reference*
- "-qdigraph" on page 124
- "-qlanglvl" on page 188

-qtune

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Tunes instruction selection, scheduling, and other architecture-dependent performance enhancements to run best on a specific hardware architecture.

Syntax

```
               ┌─balanced─┐
▶▶── -q─tune──=──┼─auto─────┼──────────────────────────────────────▶◀
               ├─ppc970───┤
               ├─pwr3─────┤
               ├─pwr4─────┤
               ├─pwr5─────┤
               ├─pwr6─────┤
               ├─pwr7─────┤
               ├─rs64b────┤
               └─rs64c────┘
```
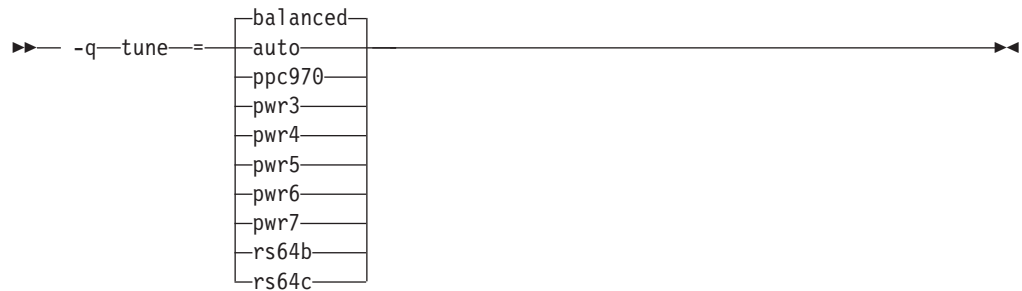
Defaults

-qtune=balanced when the default **-qarch** setting is in effect. Otherwise, the default depends on the effective **-qarch** setting. See Table 28 on page 310 for details.

Parameters

auto
Optimizations are tuned for the platform on which the application is compiled.

balanced
Optimizations are tuned across a selected range of recent hardware.

ppc970
Optimizations are tuned for the PowerPC 970 processor.

pwr3
Optimizations are tuned for the POWER3 hardware platforms.

pwr4
Optimizations are tuned for the POWER4 hardware platforms.

pwr5
Optimizations are tuned for the POWER5 hardware platforms.

pwr6
Optimizations are tuned for the POWER6 hardware platforms.

pwr7
Optimizations are tuned for the POWER7 hardware platforms.

rs64b
Optimizations are tuned for the RS64II processor.

rs64c
Optimizations are tuned for the RS64III processor.

Usage

If you want your program to run on more than one architecture, but to be tuned to a particular architecture, you can use a combination of the **-qarch** and **-qtune** options. These options are primarily of benefit for floating-point intensive programs.

By arranging (scheduling) the generated machine instructions to take maximum advantage of hardware features such as cache size and pipelining, **-qtune** can improve performance. It only has an effect when used in combination with options that enable optimization.

Although changing the **-qtune** setting may affect the performance of the resulting executable, it has no effect on whether the executable can be executed correctly on a particular hardware platform.

Acceptable combinations of **-qarch** and **-qtune** are shown in the following table.

*Table 28. Acceptable **-qarch/-qtune** combinations*

-qarch option	Default -qtune setting	Available -qtune settings
ppc	balanced	auto | rs64b | rs64c | pwr3 | pwr4 | pwr5 | pwr6 | pwr7 | ppc970 | balanced
ppcgr	balanced	auto | rs64b | rs64c | pwr3 | pwr4 | pwr5 | pwr6 | pwr7 | ppc970 | balanced
ppc64	balanced	auto | | rs64b | rs64c | pwr3 | pwr4 | pwr5 | pwr6 | pwr7 | ppc970 | balanced
ppc64gr	balanced	auto | rs64b | rs64c | pwr3 | pwr4 | pwr5 | pwr6 | pwr7 | ppc970 | balanced
ppc64grsq	balanced	auto | rs64b | rs64c | pwr3 | pwr4 | pwr5 | pwr6 | pwr7 | ppc970 | balanced
ppc64v	ppc970	auto | ppc970 | pwr6 | balanced
ppc970	ppc970	auto | ppc970 | balanced
pwr3	pwr3	auto | pwr3 | pwr4 | pwr5 | pwr7 | ppc970 | balanced
pwr4	pwr4	auto | pwr4 | pwr5 | pwr7 | ppc970 | balanced
pwr5	pwr5	auto | pwr5 | pwr7 | balanced
pwr5x	pwr5	auto | pwr5 | pwr7 | balanced
pwr6	pwr6	auto | pwr6 | pwr7 | balanced
pwr6e	pwr6	auto | pwr6 | balanced
pwr7	pwr7	auto | pwr7 | balanced
rs64b	rs64b	auto | rs64b
rs64c	rs64c	auto | rs64c

Predefined macros

None.

Examples

To specify that the executable program testing compiled from myprogram.c is to be optimized for a POWER7 hardware platform, enter:

```
xlc -o testing myprogram.c -qtune=pwr7
```

Related information
* "-qarch" on page 95
* "-q32, -q64" on page 87
* "Specifying compiler options for architecture-specific, 32-bit or 64-bit compilation" on page 9
* "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*

-U

Category

Language element control

Pragma equivalent

None.

Purpose

Undefines a macro defined by the compiler or by the **-D** compiler option.

Syntax

```
►►─── -U──name────────────────────────────────────────────────►◄
```

Defaults

Many macros are predefined by the compiler; see Chapter 6, "Compiler predefined macros," on page 391 for those that can be undefined (that is, are not *protected*). The compiler configuration file also uses the **-D** option to predefine several macro names for specific invocation commands; for details, see the configuration file for your system.

Parameters

name
    The macro you want to undefine.

Usage

The **-U** option is *not* equivalent to the #undef preprocessor directive. It *cannot* undefine names defined in the source by the #define preprocessor directive. It can only undefine names defined by the compiler or by the **-D** option.

The **-U***name* option has a higher precedence than the **-D***name* option.

Predefined macros

None.

Examples

Assume that your operating system defines the name __unix, but you do not want your compilation to enter code segments conditional on that name being defined, compile myprogram.c so that the definition of the name __unix is nullified by entering:

```
xlc myprogram.c  -U__unix
```

Related information
• "-D" on page 120

-qunroll

Category

Optimization and tuning

Pragma equivalent

#pragma options [no]unroll, #pragma unroll

Purpose

Controls loop unrolling, for improved performance.

When **unroll** is in effect, the optimizer determines and applies the best unrolling factor for each loop; in some cases, the loop control may be modified to avoid unnecessary branching. The compiler remains the final arbiter of whether the loop is actually unrolled. You can use the **#pragma unroll** directive to gain more control over unrolling.

Syntax

Option syntax
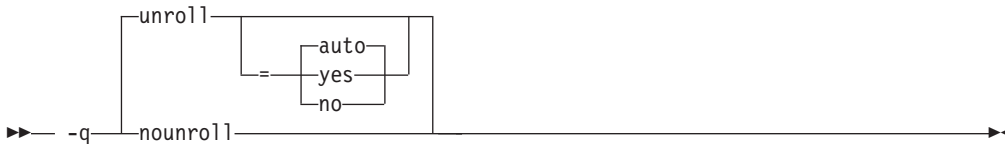
Pragma syntax
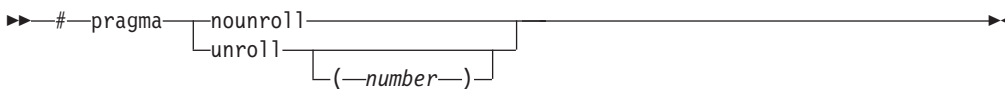
Defaults

-qunroll=auto

Parameters

auto (option only)
> Instructs the compiler to perform basic loop unrolling.

yes (option only)
> Instructs the compiler to search for more opportunities for loop unrolling than that performed with **auto**. In general, this suboption has more chances to increase compile time or program size than **auto** processing, but it may also improve your application's performance.

no (option only)
> Instructs the compiler to not unroll loops.

number **(pragma only)**
> Forces *number* - 1 replications of the designated loop body or full unrolling of the loop, whichever occurs first. The value of *number* is unbounded and must

be a positive integer. Specifying **#pragma unroll(1)** effectively disables loop unrolling, and is equivalent to specifying **#pragma nounroll**. If *number* is not specified and if **-qhot**, **-qsmp**, or **-O4** or higher is specified, the optimizer determines an appropriate unrolling factor for each nested loop.

Specifying **-qunroll** without any suboptions is equivalent to **-qunroll=yes**.

-qnounroll is equivalent to **-qunroll=no**.

Usage

The pragma overrides the **-q[no]unroll** compiler option setting for a designated loop. However, even if **#pragma unroll** is specified for a given loop, the compiler remains the final arbiter of whether the loop is actually unrolled.

Only one pragma may be specified on a loop. The pragma must appear immediately before the loop or the **#pragma block_loop** directive to have effect.

The pragma affects only the loop that follows it. An inner nested loop requires a **#pragma unroll** directive to precede it if the desired loop unrolling strategy is different from that of the prevailing **-q[no]unroll** option.

The **#pragma unroll** and **#pragma nounroll** directives can only be used on `for` loops or **#pragma block_loop** directives. They cannot be applied to `do while` and `while` loops.

The loop structure must meet the following conditions:
- There must be only one loop counter variable, one increment point for that variable, and one termination variable. These cannot be altered at any point in the loop nest.
- Loops cannot have multiple entry and exit points. The loop termination must be the only means to exit the loop.
- Dependencies in the loop must not be "backwards-looking". For example, a statement such as `A[i][j] = A`**`[i -1]`**`[j + 1] + 4)` must not appear within the loop.

Predefined macros

None.

Examples

In the following example, the **#pragma unroll(3)** directive on the first `for` loop requires the compiler to replicate the body of the loop three times. The **#pragma unroll** on the second `for` loop allows the compiler to decide whether to perform unrolling.

```
#pragma unroll(3)
for( i=0;i < n; i++)
{
     a[i] = b[i] * c[i];
}

#pragma unroll
for( j=0;j < n; j++)
```

```
{
    a[j] = b[j] * c[j];

}
```

In this example, the first **#pragma unroll(3)** directive results in:

```
i=0;
if (i>n-2) goto remainder;
for (; i<n-2; i+=3) {
  a[i]=b[i] * c[i];
  a[i+1]=b[i+1] * c[i+1];
  a[i+2]=b[i+2] * c[i+2];
}
if (i<n) {
  remainder:
  for (; i<n; i++) {
    a[i]=b[i] * c[i];
  }
}
```

Related information
- "#pragma block_loop" on page 335
- "#pragma loopid" on page 351
- "#pragma stream_unroll" on page 370
- "#pragma unrollandfuse" on page 372

-qunwind
Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Specifies whether the call stack can be unwound by code looking through the saved registers on the stack.

Specifying **-qnounwind** asserts to the compiler that the stack will not be unwound, and can improve optimization of non-volatile register saves and restores.

Syntax

```
          ┌─unwind──┐
►►─── -q───┴─nounwind─┴───────────────────────────────────────►◄
```

Defaults

-qunwind

Usage

The setjmp and longjmp families of library functions are safe to use with **-qnounwind**.

▶ `C++` Specifying **-qnounwind** also implies **-qnoeh**.

Predefined macros

None.

Related information
- "-qeh (C++ only)" on page 129

-qupconv (C only)
Category

Portability and migration

Pragma equivalent

#pragma options [no]upconv

Purpose

Specifies whether the `unsigned` specification is preserved when integral promotions are performed.

When **noupconv** is in effect, any `unsigned` type smaller than an `int` is converted to `int` during integral promotions. When **upconv** is in effect, these types are converted to `unsigned int` during integral promotions.

Syntax

```
                  ┌─noupconv─┐
►►─── -q──────────┴─upconv───┴──────────────────────────────────────►◄
```

Defaults
- **-qnoupconv** for all language levels except **classic** or **extended**
- **-qupconv** when the **classic** or **extended** language levels are in effect

Usage

Sign preservation is provided for compatibility with older dialects of C. The ANSI C standard requires value preservation as opposed to sign preservation.

Predefined macros

None.

Examples

To compile `myprogram.c` so that all `unsigned` types smaller than `int` are converted to `unsigned int`, enter:

```
xlc myprogram.c -qupconv
```

The following short listing demonstrates the effect of **-qupconv**:

```
#include <stdio.h>
int main(void) {
  unsigned char zero = 0;
  if (-1 <zero)
    printf("Value-preserving rules in effect\n");
  else
    printf("Unsignedness-preserving rules in effect\n");
  return 0;
}
```

Related information
- "Usual arithmetic conversions" in the *XL C/C++ Language Reference*
- "-qlanglvl" on page 188

-qutf

Category

Language element control

Pragma equivalent

None.

Purpose

Enables recognition of UTF literal syntax.

Syntax

```
►►── -q──┬─noutf─┬───────────────────────────────────────────◄►
         └─utf───┘
```

Defaults
- ▶ C ◀ -qnoutf

- ▶ C++ ◀ -qutf

Usage

The compiler uses **iconv** to convert the source file to Unicode. If the source file cannot be converted, the compiler will ignore the **-qutf** option and issue a warning.

Predefined macros

None.

Related information
- "UTF literals" in the *XL C/C++ Language Reference*

-v, -V

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Reports the progress of compilation, by naming the programs being invoked and the options being specified to each program.

When the **-v** option is in effect, information is displayed in a comma-separated list. When the **-V** option is in effect, information is displayed in a space-separated list.

Syntax

```
►►──┬─ -v ─┬──────────────────────────────────────────────────────────►◄
    └─ -V ─┘
```

Defaults

The compiler does not display the progress of the compilation.

Usage

The **-v** and **-V** options are overridden by the **-#** option.

Predefined macros

None.

Examples

To compile `myprogram.c` so you can watch the progress of the compilation and see messages that describe the progress of the compilation, the programs being invoked, and the options being specified, enter:

```
xlc myprogram.c -v
```

Related information
- "-# (pound sign)" on page 86

-qversion

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Displays the version and release of the compiler being invoked.

Syntax

```
          ┌─noversion─┐
►►── -q────┤─version──┼──────────────────────────►◄
          │          │
          └──=──verbose─┘
```

Defaults

-qnoversion

Parameters

verbose
   Additionally displays information about the version, release, and level of each
   compiler component installed.

Usage

When you specify **-qversion**, the compiler displays the version information and
exits; compilation is stopped

-qversion specified without the **verbose** suboption shows compiler information in
the format:

*product_name*Version: *VV.RR.MMMM.LLLL*

where:
V   Represents the version.
R   Represents the release.
M   Represents the modification.
L   Represents the level.

Example:

```
IBM XL C/C++ for Linux, V10.1
Version: 10.01.0000.0001
```

-qversion=verbose shows component information in the following format:

component_name Version: *VV.RR*(*product_name*) Level: *component_level*

where:
component_name
         Specifies an installed component, such as the low-level optimizer.
component_level
         Represents the level of the installed component.

Example:

```
IBM XL C/C++ for Linux, V10.1
Version: 10.01.0000.0001
Driver Version: 10.01(C/C++) Level: 060414
C Front End Version: 10.01(C/C++) Level: 060419
C++ Front End Version: 10.01(C/C++) Level: 060420
High Level Optimizer Version: 10.01(C/C++) and 12.01(Fortran) Level: 060411
Low Level Optimizer Version: 10.01(C/C++) and 12.01(Fortran) Level: 060418
```

If you want to save this information to the output object file, you can do so with
the **-qsaveopt -c** options.

Predefined macros

None.

Related information

- "-qsaveopt" on page 263

-qvrsave

Category

Object code control

Pragma equivalent

#pragma altivec_vrsave

Purpose

Enables code in function prologs and epilogs to maintain the VRSAVE register.

Each bit in the VRSAVE register corresponds to a vector register and, if set to 1, indicates that the corresponding vector register contains data to be saved when a context switch occurs. Use **-qvrsave** to indicate to the compiler that functions in the compilation unit include code needed to maintain the VRSAVE register. Use **-qnovrsave** to indicate to the compiler that functions in the compilation unit do not include code needed to maintain the VRSAVE register.

You can use the pragma to override the current setting of the compiler option for individual functions within your program source.

Syntax

Option syntax

```
           ┌─vrsave──┐
►►── -q──┴─novrsave─┴──────────────────────────────────────────►◄
```

Pragma syntax

```
                                  ┌─on───┐
►►──#──pragma──altivec_vrsave──┼─off──┼───────────────────────►◄
                                  └─allon─┘
```

Defaults

vrsave: The VRSAVE register is always maintained.

Parameters

on (pragma only)
Function prologs and epilogs include code to maintain the VRSAVE register.

off (pragma only)
Function prologs and epilogs do not include code to maintain the VRSAVE register.

allon (pragma only)

The function containing pragma sets all bits of the VRSAVE register to 1, indicating that all vectors are used and should be saved if a context switch occurs.

Usage

This option and pragma are only supported when **-qaltivec** is in effect.

The pragma can be used only within a function, and its effects apply only to the function in which it appears. Specifying this pragma with different settings within the same function will create an error condition.

Predefined macros

None.

Related information
• "-qaltivec" on page 94

-w

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Suppresses informational, language-level and warning messages.

▶ C ◀ This option is equivalent to specifying **-qflag=e : e**. ▶ C++ ◀ This option is equivalent to specifying **-qflag=s : s**.

Syntax

▶▶── -w ──◀◀

Defaults

All informational and warning messages are reported.

Usage

Informational and warning messages that supply additional information to a severe error are not disabled by this option.

Predefined macros

None.

Examples

To compile `myprogram.c` so that no warning messages are displayed, enter:

```
xlc myprogram.c -w
```

The following example shows how informational messages that result from a
severe error, in this case caused by problems with overload resolution in C++, are
not disabled :

```
void func(int a){}
void func(int a, int b){}
int main(void)
{
func(1,2,3);
return 0;
}
```

The output is as follows:

```
"x.cpp", line 6.4: 1540-0218 (S) The call does not match any parameter list for
                                "func".
"x.cpp", line 1.6: 1540-1283 (I) "func(int)" is not a viable candidate.
"x.cpp", line 6.4: 1540-0215 (I) The wrong number of arguments have been
                                specified for "func(int)".
"x.cpp", line 2.6: 1540-1283 (I) "func(int, int)" is not a viable candidate.
"x.cpp", line 6.4: 1540-0215 (I) The wrong number of arguments have been specified
                                for "func(int, int)".
```

Related information
- "-qflag" on page 136
- "-qsuppress" on page 291

-W

Category

Compiler customization

Pragma equivalent

None.

Purpose

Passes the listed options to a component that is executed during compilation.

Syntax
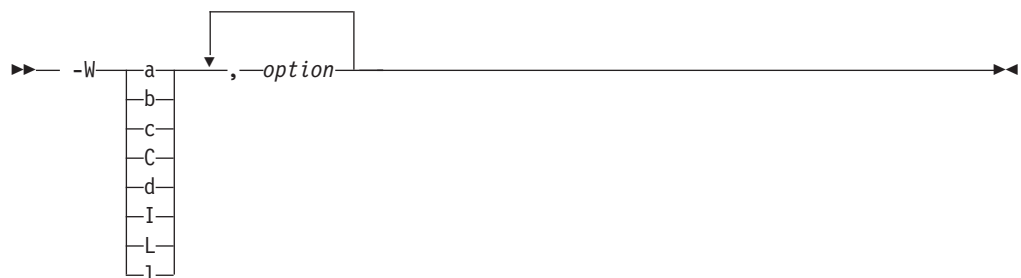
Parameters

option
> Any option that is valid for the component to which it is being passed. Spaces must not appear before the *option*.

The following table shows the correspondence between **-W** parameters and the component executable names:

Parameter	Description	Executable name
a	Assembler	as
b	Low-level optimizer	xlCcode
c	Compiler front end	xlcentry, xlCentry
▶ C++ C	C++ compiler front end	xlCentry
d	Disassembler	dis
I	High-level optimizer, compile step	ipa
L	High-level optimizer, link step	ipa
l	Linker	ld

Usage

In the string following the **-W** option, use a comma as the separator for each option, and do not include any spaces. If you need to include a character that is special to the shell in the option string, precede the character with a backslash. For example, if you use the **-W** option in the configuration file, you can use the escape sequence backslash comma (\,) to represent a comma in the parameter string.

You do not need the **-W** option to pass most options to the linker **ld**; unrecognized command-line options, except **-q** options, are passed to it automatically. Only linker options with the same letters as compiler options, such as **-v** or **-S**, strictly require **-W**.

Predefined macros

None.

Examples

To compile the file `file.c` and pass the linker option **-berok** to the linker, enter the following command:

```
xlc -Wl,-berok file.c
```

To compile the file `uses_many_symbols.c` and the assembly file `produces_warnings.s` so that `produces_warnings.s` is assembled with the assembler option **-x** (issue warnings and produce cross-reference), and the object files are linked with the option **-s** (write list of object files and strip final executable file), issue the following command:.

```
xlc -Wa,-x -Wl,-s produces_warnings.s uses_many_symbols.c
```

Related information
- "Invoking the compiler" on page 1

-qwarn0x (C++0x)

Note: C++0x is a new version of the C++ programming language standard. This is a draft standard and has not been officially adopted in its entirety. Note that future levels of support for this standard are likely to change. The implementation of the language level is based on IBM's interpretation of the draft C++0x standard, and is subject to change at any time without notice. IBM makes no attempt to maintain compatibility with earlier releases, in source or binary, of the new C++0x **-qlanglvl** suboptions (their names or their semantics) and therefore they should not be relied on as a stable programming interface.

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Controls whether to inform users with messages about differences in their programs caused by migration from the C++98 standard to the C++0x standard.

For example, when **-qlanglvl=noc99preprocessor** and **-qwarn0x** are specified, the C++0x preprocessor evaluates the controlling expressions in the `#if` and `#elif` conditional inclusion directives, and compare the evaluation results against that of the non-C++0x preprocessor. If they are different, the compiler issues the following warning message:

```
The preprocessor controlling expression evaluates differently between C++0x
and non-C++0x langlvls.
```

For another example, when you specify both the **-qlanglvl=noc99longlong** and **-qwarn0x** options, the compiler might display messages to indicate that the types of an integer literal are different between the non-C++0x and C++0x language levels. In 32-bit mode, when you use the integer literal 2147483648 to initialize a variable, the compiler issues the following message:

```
Integral constant "2147483648" has implied type unsigned long int under the non-C++0x
language levels. It has implied type long long int under C++0x.
```

The compiler displays a similar message for the literal 10000000000000000000 with the same option setting:

```
Integral constant "10000000000000000000" has implied type unsigned long long or is not allowed
with "-qlanglvl=extendedintegersafe" under C++0x. Its impiled type is not unsigned long long under
non-C++0x language levels.
```

Syntax

```
          ┌─nowarn0x─┐
►►── -q────┼─warn0x───┼──────────────────────────────────────────────►◄
```

Defaults

-qnowarn0x

Usage

This option is in effect when **-qwarn0x** is set.

Predefined macros

None.

Related information
- "-qlanglvl" on page 188

-qwarn64

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Enables checking for possible data conversion problems between 32-bit and 64-bit compiler modes.

When **-qwarn64** is in effect, informational messages are displayed where data conversion may cause problems in 64-bit compilation mode, such as:
- Truncation due to explicit or implicit conversion of `long` types into `int` types
- Unexpected results due to explicit or implicit conversion of `int` types into `long` types
- Invalid memory references due to explicit conversion by cast operations of pointer types into `int` types
- Invalid memory references due to explicit conversion by cast operations of `int` types into pointer types
- Problems due to explicit or implicit conversion of constants into `long` types
- Problems due to explicit or implicit conversion by cast operations of constants into pointer types

Syntax

```
               ┌─nowarn64─┐
►►── -q────────┼─warn64───┤────────────────────────────────►◄
```

Defaults

-qnowarn64

Usage

This option functions in either 32-bit or 64-bit compiler modes. In 32-bit mode, it functions as a preview aid to discover possible 32-bit to 64-bit migration problems.

Predefined macros

None.

Related information
- -q32, -q64
- "Compiler messages" on page 16

-qxcall

Category

Object code control

Pragma equivalent

None.

Purpose

Generates code to treat static functions within a compilation unit as if they were external functions.

Syntax

```
►►── -q──┬─noxcall─┬──────────────────────────────────────────────►◄
         └─xcall───┘
```

Defaults

-qnoxcall

Usage

-qxcall generates slower code than **-qnoxcall**.

Predefined macros

None.

Examples

To compile myprogram.c so that all static functions are compiled as external functions, enter:

```
xlc myprogram.c -qxcall
```

-qxref

Category

Listings, messages, and compiler information

Pragma equivalent

#pragma options [no]xref

Purpose

Produces a compiler listing that includes the cross-reference component of the attribute and cross-reference section of the listing.

When **xref** is in effect, a listing file is generated with a .lst suffix for each source file named on the command line. For details of the contents of the listing file, see "Compiler listings" on page 19.

Syntax

```
►►── -q──┬─noxref─────────────┬──────────────────────────────────────►◄
         └─xref─┬───────────┬─┘
                └──=──full──┘
```

Defaults

-qnoxref

Parameters

full
> Reports all identifiers in the program. If you specify **xref** without this suboption, only those identifiers that are used are reported.

Usage

A typical cross-reference listing has the form:

```
Identifier name           Description of the item
      ┌───────────┐      ┌────────────────────────┐
           xy             auto int in function adder
                         0-59Y   0-36.12Z    0-48.12Z
                                                  │││└─ Function invocation
                                                  ││└── Column number
                                                  │└─── Line number
                                                  └──── File
                                            ──────── Function definition
```

The listing uses the following character codes:

Table 29. Cross-reference listing codes

Character	Meaning
X	Function is declared.
Y	Function is defined.
Z	Function is called.
$	Type is defined, variable is declared/defined.
#	Variable is assigned to.
&	Variable is defined and initialized.
[blank]	Identifier is referenced.

The **-qnoprint** option overrides this option.

Any function defined with the **#pragma mc_func** directive is listed as being defined on the line of the pragma directive.

Predefined macros

None.

Examples

To compile `myprogram.c` and produce a cross-reference listing of all identifiers, whether they are used or not, enter:

```
xlc myprogram.c -qxref=full
```

Related information
- "-qattr" on page 102
- "#pragma mc_func" on page 354

-y

Category

Floating-point and integer control

Pragma equivalent

None.

Purpose

Specifies the rounding mode for the compiler to use when evaluating constant floating-point expressions at compile time.

Syntax

Defaults

-yn, -ydn

Parameters

The following suboptions are valid for binary floating-point types only:

m   Round toward minus infinity.

n   Round to the nearest representable number, ties to even.

p   Round toward plus infinity.

z   Round toward zero.

Usage

If your program contains operations involving long doubles, the rounding mode must be set to **-yn** (round-to-nearest representable number, ties to even).

Predefined macros

None.

Examples

To compile `myprogram.c` so that constant floating-point expressions are rounded toward zero at compile time, enter:

```
xlc myprogram.c -yz
```

Chapter 5. Compiler pragmas reference

The following sections describe the pragmas available:
- "Pragma directive syntax"
- "Scope of pragma directives" on page 330
- "Summary of compiler pragmas by functional category" on page 330
- "Individual pragma descriptions" on page 335

Pragma directive syntax

XL C/C++ supports three forms of pragma directives:

#pragma options *option_name*
> These pragmas use exactly the same syntax as their command-line option equivalent. The exact syntax and list of supported pragmas of this type are provided in "#pragma options" on page 356.

#pragma *name*
> This form uses the following syntax:

> The *name* is the pragma directive name, and the *suboptions* are any required or optional suboptions that can be specified for the pragma, where applicable.

_Pragma ("*name***")**
> This form uses the following syntax:

> For example, the statement:
> ```
> _Pragma ( "pack(1)" )
> ```
>
> is equivalent to:
> ```
> #pragma pack(1)
> ```

For all forms of pragma statements, you can specify more than one *name* and *suboptions* in a single **#pragma** statement.

The *name* on a pragma is subject to macro substitutions, unless otherwise stated. The compiler ignores unrecognized pragmas, issuing an informational message indicating this.

If you have any pragmas that are not common to both C and C++ in code that will be compiled by both compilers, you may add conditional compilation directives around the pragmas. (This is not strictly necessary since unrecognized pragmas are

ignored.) For example, **#pragma object_model** is only recognized by the C++
compiler, so you may decide to add conditional compilation directives around the
pragma.

```
#ifdef __cplusplus
#pragma object_model(pop)
#endif
```

Scope of pragma directives

Many pragma directives can be specified at any point within the source code in a
compilation unit; others must be specified before any other directives or source
code statements. In the individual descriptions for each pragma, the "Usage"
section describes any constraints on the pragma's placement.

In general, if you specify a pragma directive before any code in your source
program, it applies to the entire compilation unit, including any header files that
are included. For a directive that can appear anywhere in your source code, it
applies from the point at which it is specified, until the end of the compilation
unit.

You can further restrict the scope of a pragma's application by using
complementary pairs of pragma directives around a selected section of code. For
example, using **#pragma options source** and **#pragma options nosource** directives
as follows requests that only the selected parts of your source code be included in
your compiler listing:

```
#pragma options source

/*  Source code between the source and nosource pragma
    options is included in the compiler listing              */

#pragma options nosource
```

Many pragmas provide "pop" or "reset" suboptions that allow you to enable and
disable pragma settings in a stack-based fashion; examples of these are provided in
the relevant pragma descriptions.

Summary of compiler pragmas by functional category

The XL C/C++ pragmas available are grouped into the following categories:
- "Language element control" on page 331
- "C++ template pragmas" on page 331
- "Floating-point and integer control" on page 331
- "Error checking and debugging" on page 331
- "Listings, messages and compiler information" on page 332
- "Optimization and tuning" on page 332
- "Object code control" on page 333
- "Portability and migration" on page 333
- "Compiler customization" on page 334
- "Deprecated directives" on page 334

For descriptions of these categories, see "Summary of compiler options by
functional category" on page 67.

Language element control

Table 30. Language element control pragmas

Pragma	Description
#pragma langlvl (C only)	Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.
"#pragma mc_func" on page 354	Allows you to embed a short sequence of machine instructions "inline" within your program source code.
"#pragma options" on page 356	Specifies compiler options in your source program.

C++ template pragmas

Table 31. C++ template pragmas

Pragma	Description
"#pragma define, #pragma instantiate (C++ only)" on page 340	Provides an alternative method for explicitly instantiating a template class.
"#pragma do_not_instantiate (C++ only)" on page 342	Prevents the specified template declaration from being instantiated.
"#pragma implementation (C++ only)" on page 348	For use with the **-qtempinc** compiler option, supplies the name of the file containing the template definitions corresponding to the template declarations contained in a header file.

Floating-point and integer control

Table 32. Floating-point and integer control pragmas

Pragma	Description
#pragma chars	Determines whether all variables of type char are treated as either signed or unsigned.
#pragma enum	Specifies the amount of storage occupied by enumerations.

Error checking and debugging

Table 33. Error checking and debugging pragmas

Pragma	Description
"#pragma ibm snapshot" on page 347	Specifies a location at which a breakpoint can be set and defines a list of variables that can be examined when program execution reaches that location.
#pragma info	Produces or suppresses groups of informational messages.

Listings, messages and compiler information

Table 34. Listings, messages and compiler information pragmas

Pragma	Description
"#pragma report (C++ only)" on page 368	Controls the generation of diagnostic messages.

Optimization and tuning

Table 35. Optimization and tuning pragmas

Pragma	Description
"#pragma block_loop" on page 335	Marks a block with a scope-unique identifier.
"#pragma STDC cx_limited_range" on page 369	Instructs the compiler that complex division and absolute value are only invoked with values such that intermediate calculation will not overflow or lose significance.
"#pragma disjoint" on page 341	Lists identifiers that are not aliased to each other within the scope of their use.
"#pragma execution_frequency" on page 343	Marks program source code that you expect will be either very frequently or very infrequently executed.
"#pragma expected_value" on page 345	Specifies the value that a parameter passed in a function call is most likely to take at run time. The compiler can use this information to perform certain optimizations, such as function cloning and inlining.
#pragma isolated_call	Specifies functions in the source file that have no side effects other than those implied by their parameters.
"#pragma leaves" on page 350	Informs the compiler that a named function never returns to the instruction following a call to that function.
"#pragma loopid" on page 351	Marks a block with a scope-unique identifier.
#pragma nosimd	When used with **-qsimd=noauto**, disables the generation of SIMD instructions for the next loop.
#pragma novector	When used with **-qhot=novector**, disables auto-vectorization of the next loop.
"#pragma option_override" on page 358	Allows you to specify optimization options at the subprogram level that override optimization options given on the command line.
"#pragma reachable" on page 366	Informs the compiler that the point in the program after a named function can be the target of a branch from some unknown location.
"#pragma reg_killed_by" on page 367	Specifies registers that may be altered by functions specified by **#pragma mc_func**.
"#pragma stream_unroll" on page 370	When optimization is enabled, breaks a stream contained in a for loop into multiple streams.

Table 35. Optimization and tuning pragmas (continued)

Pragma	Description
#pragma unroll	Controls loop unrolling, for improved performance.
"#pragma unrollandfuse" on page 372	Instructs the compiler to attempt an unroll and fuse operation on nested for loops.

Object code control

Table 36. Object code control pragmas

Pragma	Description
#pragma alloca (C only)	Provides an inline definition of system function alloca when it is called from source code that does not include the alloca.h header.
#pragma altivec_vrsave	Enables code in function prologs and epilogs to maintain the VRSAVE register.
"#pragma comment" on page 339	Places a comment into the object module.
"#pragma hashome (C++ only)" on page 346	Informs the compiler that the specified class has a home module that will be specified by **#pragma ishome**.
"#pragma ishome (C++ only)" on page 349	Informs the compiler that the specified class's home module is the current compilation unit.
"#pragma map" on page 351	Converts all references to an identifier to another, externally defined identifier.
"#pragma pack" on page 360	Sets the alignment of all aggregate members to a specified byte boundary.
#pragma priority (C++ only)	Specifies the priority level for the initialization of static objects.
"#pragma reg_killed_by" on page 367	Specifies registers that may be altered by functions specified by **#pragma mc_func**.
#pragma strings	Specifies the storage type for string literals.
"#pragma weak" on page 374	Prevents the linker from issuing error messages if it encounters a symbol multiply-defined during linking, or if it does not find a definition for a symbol.

Portability and migration

Table 37. Portability and migration pragmas

Pragma	Description
#pragma align	Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data.

Deprecated directives

The SMP directives listed in the following table have been deprecated and might be removed in the future release. Use the corresponding OpenMP directives to obtain the same behavior.

Table 38. Deprecated SMP directives

SMP directive name	OpenMP directive/clause name
#pragma ibm critical	"#pragma omp critical" on page 387
#pragma ibm parallel_loop	The "#pragma omp parallel for" on page 384 pragma with the **schedule** clause.
#pragma ibm schedule	

The following examples show how to replace the deprecated SMP directives with their corresponding OpenMP ones.

For the **critical** pragma:

```
#pragma ibm critical(lck)
{
   ...
}
```

is replaced by

```
#pragma omp critical(lck)
{
...
}
```

For the **schedule** pragma:

```
#pragma ibm parallel_loop
#pragma ibm schedule(static, 5)
for (i=0; i<N; i++)
{
   ...
}
```

is replaced by

```
#pragma omp parallel for schedule(static, 5)
for (i=0; i<N; i++)
{
   ...
}
```

Compiler customization

Table 39. Compiler customization pragmas

Pragma	Description
#pragma complexgcc	Specifies whether to use GCC parameter-passing conventions for complex data types (equivalent to enabling **-qfloat=complexgcc**).

Individual pragma descriptions

This section contains descriptions of individual pragmas available in XL C/C++.

For each pragma, the following information is given:

Category
> The functional category to which the pragma belongs is listed here.

Purpose
> This section provides a brief description of the effect of the pragma, and why you might want to use it.

Syntax
> This section provides the syntax for the pragma. For convenience, the **#pragma** *name* form of the directive is used in each case. However, it is perfectly valid to use the alternate C99-style `_Pragma` operator syntax; see "Pragma directive syntax" on page 329 for details.

Parameters
> This section describes the suboptions that are available for the pragma, where applicable.

Usage This section describes any rules or usage considerations you should be aware of when using the pragma. These can include restrictions on the pragma's applicability, valid placement of the pragma, and so on.

Examples
> Where appropriate, examples of pragma directive use are provided in this section.

#pragma align

See "-qalign" on page 91.

#pragma alloca (C only)

See "-qalloca, -ma (C only)" on page 93.

#pragma altivec_vr_save

See "-qvrsave" on page 319.

#pragma block_loop

Category

Optimization and tuning

Purpose

Marks a block with a scope-unique identifier.

Syntax

```
►►─#─pragma─block_loop─(─expression─,─┬─name─┬─)──────────────►◄
                                      └──,◄──┘
```

Parameters

expression
> An integer expression representing the size of the iteration group.

name
> An identifier that is unique within the scoping unit. If you do not specify a *name*, blocking occurs on the first `for` loop or loop following the **#pragma block_loop** directive.

Usage

For loop blocking to occur, a **#pragma block_loop** directive must precede a `for` loop.

If you specify **#pragma unroll**, **#pragma unrollandfuse** or **#pragma stream_unroll** for a blocking loop, the blocking loop is unrolled, unrolled and fused or stream unrolled respectively, if the blocking loop is actually created. Otherwise, this directive has no effect.

If you specify **#pragma unrollandfuse**, **#pragma unroll** or **#pragma stream_unroll** directive for a blocked loop, the directive is applied to the blocked loop after the blocking loop is created. If the blocking loop is not created, this directive is applied to the loop intended for blocking, as if the corresponding **#pragma block_loop** directive was not specified.

You must not specify **#pragma block_loop** more than once, or combine the directive with **#pragma nounroll**, **#pragma unroll**, **#pragma nounrollandfuse**, **#pragma unrollandfuse**, or **#pragma stream_unroll** directives for the same `for` loop. Also, you should not apply more than one **#pragma unroll** directive to a single block loop directive.

Processing of all **#pragma block_loop** directives is always completed before performing any unrolling indicated by any of the unroll directives

Examples

The following two examples show the use of **#pragma block_loop** and **#pragma loop_id** for loop tiling:

```
#pragma block_loop(50, mymainloop)
#pragma block_loop(20, myfirstloop, mysecondloop)
#pragma loopid(mymainloop)
  for (i=0; i < n; i++)
  {
#pragma loopid(myfirstloop)
    for (j=0; j < m; j++)
    {
#pragma loopid(mysecondloop)
      for (k=0; k < m; k++)
      {
        ...
      }
    }
  }

#pragma block_loop(50, mymainloop)
#pragma block_loop(20, myfirstloop, mysecondloop)
#pragma loopid(mymainloop)
      for (i=0; i < n; n++)
      {
#pragma loopid(myfirstloop)
```

```
                    for (j=0; j < m; j++)
                    {
#pragma loopid(mysecondloop)
                        for (k=0; k < m; k++)
                        {
                                ...
                        }
                    }
            }
```

The following example shows the use **#pragma block_loop** and **#pragma loop_id** for loop interchange.

```
        for (i=0; i < n; i++)
        {
                for (j=0; j < n; j++)
                {
#pragma block_loop(1,myloop1)
                    for (k=0; k < m; k++)
                    {
#pragma loopid(myloop1)
                            for (l=0; l < m; l++)
                            {
                                    ...
                            }
                    }
                }
        }
```

The following example shows the use of **#pragma block_loop** and **#pragma loop_id** for loop tiling for multi-level memory hierarchy:

```
#pragma block_loop(l3factor, first_level_blocking)
  for (i=0; i < n; i++)
  {
#pragma loopid(first_level_blocking)
#pragma block_loop(l2factor, inner_space)
    for (j=0; j < n; j++)
    {
#pragma loopid(inner_space)
      for (k=0; k < m; k++)
      {
        for (l=0; l < m; l++)
        {
          ...
        }
      }
    }
  }
```

The following example uses **#pragma unrollandfuse** and **#pragma block_loop** to unroll and fuse a blocking loop.

```
#pragma unrollandfuse
#pragma block_loop(10)
   for (i = 0; i < N; ++i) {
   }
```

In this case, if the block loop directive is ignored, the unroll directives have no effect.

The following example shows the use of **#pragma unroll** and **#pragma block_loop** to unroll a blocked loop.

```
#pragma block_loop(10)
#pragma unroll(2)
  for (i = 0; i < N; ++i) {
  }
```

In this case, if the block loop directive is ignored, the unblocked loop is still subjected to unrolling. If blocking does happen, the unroll directive is applied to the blocked loop.

The following examples show invalid uses of the directive. The first example shows **#pragma block_loop** used on an undefined loop identifier:

```
#pragma block_loop(50, myloop)
  for (i=0; i < n; i++)
  {
  }
```

Referencing `myloop` is not allowed, since it is not in the nest and may not be defined.

In the following example, referencing `myloop` is not allowed, since it is not in the same loop nest:

```
  for (i=0; i < n; i++)
  {
#pragma loopid(myLoop)
    for (j=0; j < i; j++)
    {
      ...
    }
  }
#pragma block_loop(myLoop)
  for (i=0; i < n; i++)
  {
    ...
  }
```

The following examples are invalid since the unroll directives conflict with each other:

```
#pragma unrollandfuse(5)
#pragma unroll(2)
  #pragma block_loop(10)
          for (i = 0; i < N; ++i) {
        }
#pragma block_loop(10)
#pragma unroll(5)
#pragma unroll(10)
  for (i = 0; i < N; ++i) {
  }
```

Related information
- "#pragma loopid" on page 351
- "-qunroll" on page 312
- "#pragma unrollandfuse" on page 372
- "#pragma stream_unroll" on page 370

#pragma chars

See "-qchars" on page 109.

#pragma comment

Category

Object code control

Purpose

Places a comment into the object module.

Syntax

```
►►─#─pragma─comment─(──┬─compiler───────────────────────────┬─)────►◄
                      ├─date───────────────────────────────┤
                      ├─timestamp──────────────────────────┤
                      └─┬─copyright─┬──┬──────────────────────┬┘
                        └─user──────┘  └─,─"─token_sequence─"─┘
```
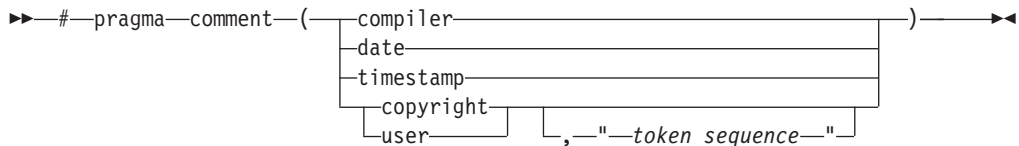
Parameters

compiler
Appends the name and version of the compiler at the end of the generated object module.

date
The date and time of the compilation are appended at the end of the generated object module.

timestamp
Appends the date and time of the last modification of the source at the end of the generated object module.

copyright
Places the text specified by the *token_sequence*, if any, into the generated object module. The *token_sequence* is included in the generated executable and loaded into memory when the program is run.

user
Places the text specified by the *token_sequence*, if any, into the generated object module. The *token_sequence* is included in the generated executable but is *not* loaded into memory when the program is run.

token_sequence
The characters in this field, if specified, must be enclosed in double quotation marks ("). If the string literal specified in the *token_sequence* exceeds 32 767 bytes, an information message is emitted and the pragma is ignored.

Usage

More than one **comment** directive can appear in a translation unit, and each type of **comment** directive can appear more than once, with the exception of **copyright**, which can appear only once.

You can display the object-file comments by using the operating system **strings** command.

Examples

Assume we have the following program code: `tt.c`:

```
#pragma comment(date)
#pragma comment(compiler)
#pragma comment(timestamp)
#pragma comment(copyright,"My copyright")
int main() { return 0; }
```

Issuing the command:

```
xlc -c tt.c
strings -a tt.o
```

will cause the comment information embedded in tt.o to be displayed, along with
any other strings that may be found in tt.o. For example, assuming the program
code shown above:

```
@.text
.data
@.bss
.comment
Thu Sep 24 16:44:25 EDT 2010IBM XL C for Linux ---- Version 11.1.0.0
Thu Sep 24 16:44:09 EDT 2010
main
My copyright
.file
tt.c
.text
.data
.bss
.main
_$STATIC
_$STATIC
main
main
Thu Sep 24 16:44:25 2010
IBM XL C for Linux, Version 11.1.0.0 ---
```

#pragma complexgcc

See "-qcomplexgccincl" on page 115.

#pragma define, #pragma instantiate (C++ only)
Category

Template control

Purpose

Provides an alternative method for explicitly instantiating a template class.

Syntax

►►—#—pragma——┬—define——┬——(—*template_class_name*—)————————————◄◄
             └—instantiate—┘

Parameters

template_class_name
    The name of the template class to be instantiated.

Usage

This pragma provides the equivalent functionality to C++ explicit instantiation definitions. It is provided for compatibility with earlier releases only. New applications should use C++ explicit instantiation definitions.

This pragma can appear anywhere an explicit instantiation definition can appear.

Examples

The following directive:

```
#pragma define(Array<char>)
```

is equivalent to the following explicit instantiation:

```
template class Array<char>;
```

Related information
- "Explicit instantiation" in the *XL C/C++ Language Reference*
- "#pragma do_not_instantiate (C++ only)" on page 342

#pragma disjoint
Category

Optimization and tuning

Purpose

Lists identifiers that are not aliased to each other within the scope of their use.

By informing the compiler that none of the identifiers listed in the pragma shares the same physical storage, the pragma provides more opportunity for optimizations.

Syntax

```
►►──#pragma disjoint────────────────────────────────────────────►

►─(─┬─────────┬─ variable_name ─┬─,─┬─────────┬─ variable_name ─)───────►◄
    └───◄──┘                     └───◄──┘
        *                            *
```

Parameters

variable_name
> The name of a variable. It must not refer to any of the following:
> - A member of a structure, class, or union
> - A structure, union, or enumeration tag
> - An enumeration constant
> - A `typedef` name

- A label

Usage

The **#pragma disjoint** directive asserts that none of the identifiers listed in the pragma share physical storage; if any the identifiers *do* actually share physical storage, the pragma may give incorrect results.

The pragma can appear anywhere in the source program that a declaration is allowed. An identifier in the directive must be visible at the point in the program where the pragma appears.

You must declare the identifiers before using them in the pragma. Your program must not dereference a pointer in the identifier list nor use it as a function argument before it appears in the directive.

This pragma can be disabled with the **-qignprag** compiler option.

Examples

The following example shows the use of **#pragma disjoint**.

```
int a, b, *ptr_a, *ptr_b;

#pragma disjoint(*ptr_a, b)    /* *ptr_a never points to b */
#pragma disjoint(*ptr_b, a)    /* *ptr_b never points to a */
one_function()
{
    b = 6;
    *ptr_a = 7;    /* Assignment will not change the value of b  */

  another_function(b);    /* Argument "b" has the value 6  */
}
```

External pointer `ptr_a` does not share storage with and never points to the external variable b. Consequently, assigning 7 to the object to which `ptr_a` points will not change the value of b. Likewise, external pointer `ptr_b` does not share storage with and never points to the external variable a. The compiler can assume that the argument to `another_function` has the value 6 and will not reload the variable from memory.

#pragma do_not_instantiate (C++ only)
Category

Template control

Purpose

Prevents the specified template declaration from being instantiated.

You can use this pragma to suppress the implicit instantiation of a template for which a definition is supplied.

Syntax

►►—#—pragma—do_not_instantiate—*template_class_name*————————————►◄

Parameters

template_class_name
    The name of the template class that should not be instantiated.

Usage

Note: C++0x is a new version of the C++ programming language standard. This is a draft standard and has not been officially adopted in its entirety. The implementation of C++0x is based on IBM's interpretation of the draft C++0x standard and is subject to change at any time without notice. IBM makes no attempt to maintain compatibility with earlier releases and therefore the C++0x language extension should not be relied on as a stable programming interface.

If you are handling template instantiations manually (that is, **-qnotempinc** and **-qnotemplateregistry** are specified), and the specified template instantiation already exists in another compilation unit, using **#pragma do_not_instantiate** ensures that you do not get multiple symbol definitions during the link step.

▶ `C++0x` **#pragma do_not_instantiate** on a class template specialization is treated as an explicit instantiation declaration of the template. This pragma provides a subset of the functionality of the explicit instantiation declarations feature, which is introduced by the C++0x standard. It is provided for compatibility purposes only and is not recommended. New applications should use explicit instantiation declarations instead.

You can also use the **-qtmplinst** option to suppress implicit instantiation of template declarations for multiple compilation units. See "-qtmplinst (C++ only)" on page 305.

Examples

The following shows the usage of the pragma:

```
#pragma do_not_instantiate Stack < int >
```

Related information
* "#pragma define, #pragma instantiate (C++ only)" on page 340
* "-qtmplinst (C++ only)" on page 305
* "Explicit instantiation" in the *XL C/C++ Language Reference*
* "-qtempinc (C++ only)" on page 297
* "-qtemplateregistry (C++ only)" on page 300

#pragma enum

See "-qenum" on page 129.

#pragma execution_frequency
Category

Optimization and tuning

Purpose

Marks program source code that you expect will be either very frequently or very infrequently executed.

When optimization is enabled, the pragma is used as a hint to the optimizer.

Syntax

```
►►─#─pragma─execution_frequency─(──┬─very_low──┬─)──────────────────────────►◄
                                   └─very_high─┘
```

Parameters

very_low
   Marks source code that you expect will be executed very infrequently.

very_high
   Marks source code that you expect will be executed very frequently.

Usage

Use this pragma in conjunction with an optimization option; if optimization is not enabled, the pragma has no effect.

The pragma must be placed within block scope, and acts on the closest point of branching.

Examples

In the following example, the pragma is used in an `if` statement block to mark code that is executed infrequently.

```
int *array = (int *) malloc(10000);

if (array == NULL) {
    /* Block A */
    #pragma execution_frequency(very_low)
    error();
}
```

In the next example, the code block `Block B` is marked as infrequently executed and `Block C` is likely to be chosen during branching.

```
if (Foo > 0) {
    #pragma execution_frequency(very_low)
    /* Block B */
    doSomething();
} else {
    /* Block C */
    doAnotherThing();
}
```

In this example, the pragma is used in a `switch` statement block to mark code that is executed frequently.

```
while (counter > 0) {
    #pragma execution_frequency(very_high)
    doSomething();
} /* This loop is very likely to be executed.    */

switch (a) {
    case 1:
        doOneThing();
        break;
    case 2:
        #pragma execution_frequency(very_high)
```

```
        doTwoThings();
        break;
    default:
        doNothing();
}    /* The second case is frequently chosen.   */
```

The following example shows how the pragma must be applied at block scope and affects the closest branching.

```
int a;
#pragma execution_frequency(very_low)
int b;

int foo(boolean boo) {
    #pragma execution_frequency(very_low)
    char c;

    if (boo) {
        /* Block A */
        doSomething();
        {
            /* Block C */
            doSomethingAgain();
            #pragma execution_frequency(very_low)
            doAnotherThing();
        }
    } else {
        /* Block B */
        doNothing();
    }

    return 0;
}

#pragma execution_frequency(very_low)
```

#pragma expected_value

Category

Optimization and tuning

Purpose

Specifies the value that a parameter passed in a function call is most likely to take at run time. The compiler can use this information to perform certain optimizations, such as function cloning and inlining.

Syntax

►►—#pragma expected_value—(—*argument*—,—*value*—)—————————————————►◄

Parameters

argument
    The name of the parameter for which you want to provide the expected value. The parameter must be of a simple built-in integral, Boolean, character, or floating-point type.

value

> A constant literal representing the value that you expect will most likely be taken by the parameter at run time. *value* can be an expression as long as it is a compile time constant expression.

Usage

The directive must appear inside the body of a function definition, before the first statement (including declaration statements). It is not supported within nested functions.

If you specify an expected value of a type different from that of the declared type of the parameter variable, the value will be implicitly converted only if allowed. Otherwise, a warning is issued.

For each parameter that will be provided the expected value there is a limit of one directive. Parameters that will not be provided the expected value do not require a directive.

Examples

The following example tells the compiler that the most likely values for parameters a and b are 1 and 0, respectively:

```
int func(int a,int b)
{
#pragma expected_value(a,1)
#pragma expected_value(b,0)
...
...
}
```

Related information

- "#pragma execution_frequency" on page 343

#pragma hashome (C++ only)
Category

Object code control

Purpose

Informs the compiler that the specified class has a home module that will be specified by **#pragma ishome**.

This class's virtual function table, along with certain inline functions, will not be generated as static. Instead, they will be referenced as externals in the compilation unit of the class in which **#pragma ishome** is specified.

Syntax

```
►►─#─pragma─hashome─(─class_name─┬──────────────┬─)─────────────────►◄
                                 └─allinlines─┘
```

Parameters

class_name

The name of a class to be referenced externally. *class_name* must be a class and it must be defined.

allinlines

Specifies that all inline functions from within *class_name* should be referenced as being external.

Usage

A warning will be produced if there is a **#pragma ishome** without a matching **#pragma hashome**.

Examples

In the following example, compiling the code samples will generate virtual function tables and the definition of `S::foo()` only for compilation unit `a.o`, but not for `b.o`. This reduces the amount of code generated for the application.

```
// a.h
struct S
{
   virtual void foo() {}

   virtual void bar();
};
```

```
// a.C
#pragma ishome(S)
#pragma hashome (S)

#include "a.h"

int main()
{
   S s;
   s.foo();
   s.bar();
}
```

```
// b.C
#pragma hashome(S)
#include "a.h"

void S::bar() {}
```

Related information
- "#pragma ishome (C++ only)" on page 349

#pragma ibm snapshot
Category

Error checking and debugging

Purpose

Specifies a location at which a breakpoint can be set and defines a list of variables that can be examined when program execution reaches that location.

You can use this pragma to facilitate debugging optimized code produced by the compiler.

Syntax

```
►►──#──pragma──ibm snapshot──(──▼──variable_name──)──────────────────►◄
                                  └─────,─────┘
```

Parameters

variable_name
　　A variable name. It must not refer to structure, class, or union members.

Usage

During a debugging session, you can place a breakpoint on the line at which the directive appears, to view the values of the named variables. When you compile with optimization and the **-g** option, the named variables are guaranteed to be visible to the debugger.

This pragma does not consistently preserve the contents of variables with a static storage class at high optimization levels. Variables specified in the directive should be considered read-only while being observed in the debugger, and should not be modified. Modifying these variables in the debugger may result in unpredictable behavior.

Examples

```
#pragma ibm snapshot(a, b, c)
```

Related information
- "-g" on page 149
- "-O, -qoptimize" on page 227

#pragma implementation (C++ only)
Category

Template control

Purpose

For use with the **-qtempinc** compiler option, supplies the name of the file containing the template definitions corresponding to the template declarations contained in a header file.

Syntax

```
►►──#──pragma──implementation──(──"──file_name──"──)────────────────────►◄
```

Parameters

file_name
> The name of the file containing the definitions for members of template classes declared in the header file.

Usage

This pragma is not normally required if your template implementation file has the same name as the header file containing the template declarations, and a .c extension. You only need to use the pragma if the template implementation file does not conform to this file-naming convention. For more information about using template implementation files, see "Using C++ templates"

#pragma implementation is only effective if the **-qtempinc** option is in effect. Otherwise, the pragma has no meaning and is ignored.

The pragma can appear in the header file containing the template declarations, or in a source file that includes the header file. It can appear anywhere that a declaration is allowed.

Related information
- "-qtempinc (C++ only)" on page 297
- "Using C++ templates"

#pragma info

See "-qinfo" on page 164.

#pragma ishome (C++ only)
Category

Object code control

Purpose

Informs the compiler that the specified class's home module is the current compilation unit.

The home module is where items, such as the virtual function table, are stored. If an item is referenced from outside of the compilation unit, it will not be generated outside its home. This can reduce the amount of code generated for the application.

Syntax

►►──#──pragma──ishome──(──*class_name*──)────────────────────►◄

Parameters

class_name
> The name of the class whose home will be the current compilation unit.

Usage

A warning will be produced if there is a **#pragma ishome** without a matching **#pragma hashome**.

Examples

See "#pragma hashome (C++ only)" on page 346

Related information
- "#pragma hashome (C++ only)" on page 346

#pragma isolated_call

See "-qisolated_call" on page 182.

#pragma langlvl (C only)

See "-qlanglvl" on page 188.

#pragma leaves
Category

Optimization and tuning

Purpose

Informs the compiler that a named function never returns to the instruction following a call to that function.

By informing the compiler that it can ignore any code after the function, the directive allows for additional opportunities for optimization.

This pragma is commonly used for custom error-handling functions, in which programs can be terminated if a certain error is encountered.

Note: The compiler automatically inserts **#pragma leaves** directives for calls to the `longjmp` family of functions (`longjmp`, `_longjmp`, `siglongjmp`, and `_siglongjmp`) when you include the setjmp.h header.

Syntax

```
>>--#--pragma--leaves--(----,----function_name----)----------><
```

Parameters

function_name
    The name of the function that does not return to the instruction following the call to it.

Defaults

Not applicable.

Examples

```
#pragma leaves(handle_error_and_quit)
void test_value(int value)
{
 if (value == ERROR_VALUE)
 {
  handle_error_and_quit(value);
  TryAgain(); // optimizer ignores this because
     // never returns to execute it
 }
}
```

Related information

- "#pragma reachable" on page 366.

#pragma loopid
Category

Optimization and tuning

Purpose

Marks a block with a scope-unique identifier.

Syntax

►►—#—pragma—loopid—(—*name*—)————————————————►◄

Parameters

name
> An identifier that is unique within the scoping unit.

Usage

The **#pragma loopid** directive must immediately precede a **#pragma block_loop** directive or `for` loop. The specified name can be used by **#pragma block_loop** to control transformations on that loop. It can also be used to provide information on loop transformations through the use of the **-qreport** compiler option.

You must not specify **#pragma loopid** more than once for a given loop.

Examples

For examples of **#pragma loopid** usage, see "#pragma block_loop" on page 335.

Related information
- "-qunroll" on page 312
- "#pragma block_loop" on page 335
- "#pragma unrollandfuse" on page 372

#pragma map
Category

Object code control

Purpose

Converts all references to an identifier to another, externally defined identifier.

Syntax

#pragma map syntax – C

►►—#—pragma—map—(—*name1*—,—"—*name2*—"—)————————◄◄

#pragma map syntax – C++

►►—#—pragma—map—(—*name1*—(—*argument_list*—)—,—"—*name2*—"—)——————◄◄

Parameters

name1

The name used in the source code. ► C *name1* can represent a data object or function with external linkage. ► C++ *name1* can represent a data object, a non-overloaded or overloaded function, or overloaded operator, with external linkage. If the name to be mapped is not in the global namespace, it must be fully qualified.

name1 should be declared in the same compilation unit in which it is referenced, but should not be defined in any other compilation unit. *name1* must not be used in another **#pragma map** directive or any assembly label declaration anywhere in the program.

► C++ *argument_list*

The list of arguments for the overloaded function or operator function designated by *name1*. If *name1* designates an overloaded function, the function must be parenthesized and must include its argument list if it exists. If *name1* designates a non-overloaded function, only *name1* is required, and the parentheses and argument list are optional.

name2

The name that will appear in the object code. ► C *name2* can represent a data object or function with external linkage.

► C++ *name2* can represent a data object, a non-overloaded or overloaded function, or overloaded operator, with external linkage. *name2* must be specified using its mangled name. To obtain C++ mangled names, compile your source to object files only, using the **-c** compiler option, and use the **nm** operating system command on the resulting object file. (See also "Name mangling" in the *XL C/C++ Language Reference* for details on using the extern "C" linkage specifier on declarations to prevent name mangling.)

If the name exceeds 65535 bytes, an informational message is emitted and the pragma is ignored.

name2 may or may not be declared in the same compilation unit in which *name1* is referenced, but must not be defined in the same compilation unit. Also, *name2* should not be referenced anywhere in the compilation unit where *name1* is referenced. *name2* must not be the same as that used in another **#pragma map** directive or any assembly label declaration in the same compilation unit.

Usage

The **#pragma map** directive can appear anywhere in the program. Note that in order for a function to be actually mapped, the map target function (*name2*) must have a definition available at link time (from another compilation unit), and the map source function (*name1*) must be called in your program.

You cannot use **#pragma map** with compiler built-in functions.

Examples

The following is an example of **#pragma map** used to map a function name (using the mangled name for the map name in C++):

```
/* Compilation unit 1: */

#include <stdio.h>

void foo();
extern void bar(); /* optional */

#if __cplusplus
#pragma map (foo, "_Z3barv")
#else#pragma map (foo, "bar")
#endif
int main()
{
foo();
}

/* Compilation unit 2: */

#include <stdio.h>

void bar()
{
printf("Hello from foo bar!\n");
}
```

The call to `foo` in compilation unit 1 resolves to a call to `bar`:

```
Hello from foo bar!
```

▶ C++ The following is an example of **#pragma map** used to map an overloaded function name (using C linkage, to avoid using the mangled name for the map name):

```
// Compilation unit 1:

#include <iostream>
#include <string>

using namespace std;

void foo();
void foo(const string&);
extern "C" void bar(const string&); // optional

#pragma map (foo(const string&), "bar")

int main()
{
foo("Have a nice day!");
}
```

```
// Compilation unit 2:

#include <iostream>
#include <string>

using namespace std;

extern "C" void bar(const string& s)
{
cout << "Hello from foo bar!" << endl;
cout << s << endl;
}
```

The call to `foo(const string&)` in compilation unit 1 resolves to a call to `bar(const string&)`:

```
Hello from foo bar!
Have a nice day!
```

Related information

- "Assembly labels" in the *XL C/C++ Language Reference*

#pragma mc_func

Category

Language element control

Purpose

Allows you to embed a short sequence of machine instructions "inline" within your program source code.

The pragma instructs the compiler to generate specified instructions in place rather than the usual linkage code. Using this pragma avoids performance penalties associated with making a call to an assembler-coded external function. This pragma is similar in function to inline `asm` statements supported in this and other compilers; see "Inline assembly statements" in the *XL C/C++ Language Reference* for more information.

Syntax

```
►►──#──pragma──mc_func──function_name──{──┬──▼──instruction_sequence──┬──}──────────►◄
                                           └────────────────────────────┘
```

Parameters

function_name
> The name of a previously-defined function containing machine instructions. If the function is not previously-defined, the compiler will treat the pragma as a function definition.

instruction_sequence
> A string containing a sequence of zero or more hexadecimal digits. The number of digits must comprise an integral multiple of 32 bits. If the string exceeds 16384 bytes, a warning message is emitted and the pragma is ignored.

Usage

This pragma defines a function and should appear in your program source only where functions are ordinarily defined.

The compiler passes parameters to the function in the same way as to any other function. For example, in functions taking integer-type arguments, the first parameter is passed to GPR3, the second to GPR4, and so on. Values returned by the function will be in GPR3 for integer values, and FPR1 for float or double values.

Code generated from *instruction_sequence* may use any and all volatile registers available on your system unless you use **#pragma reg_killed_by** to list a specific register set for use by the function. See "#pragma reg_killed_by" on page 367 for a list of volatile registers available on your system.

Inlining options do not affect functions defined by **#pragma mc_func**. However, you may be able to improve runtime performance of such functions with **#pragma isolated_call**.

Examples

In the following example, **#pragma mc_func** is used to define a function called `add_logical`. The function consists of machine instructions to add 2 integers with so-called *end-around carry*; that is, if a carry out results from the add then add the carry to the sum. This formula is frequently used in checksum computations.

```
int add_logical(int, int);
#pragma mc_func add_logical {"7c632014" "7c630194"}
                /*   addc      r3 <- r3, r4          */
                /*   addze     r3 <- r3, carry bit   */


main() {

     int i,j,k;

     i = 4;
     k = -4;
     j = add_logical(i,k);
     printf("\n\nresult = %d\n\n",j);
}
```

The result of running the program is:

```
result = 1
```

Related information
- "-qisolated_call" on page 182
- "#pragma reg_killed_by" on page 367
- "Inline assembly statements" in the *XL C/C++ Language Reference*

#pragma nofunctrace
Category

Error checking and debugging

Purpose

Disables tracing for a given function or a list of specified functions.

Syntax

```
>>--#--pragma--nofunctrace--(--,--function_name--)--------------><
```

Parameters

function_name
   The name of the function for which you want to disable tracing.

Usage

When you use **#pragma nofunctrace** to specify a list of functions for which you want to disable tracing, use parenthesis **()** and encapsulate the functions in it. For a list of functions, use a comma **,** to separate them. For example, to disable tracing for function a, use #pragma nofunctrace(a). To disable tracing for functions a, b, and c, use #pragma nofunctrace(a,b,c).

If you have two functions: foo(int) and foo(double), use #pragma nofunctrace(foo(int)) disables tracing for foo(int) but not foo(double).

Two colons in a row :: are considered scope qualifiers. For example, when you call -qfunctrace+A::B:C, the compiler traces functions that begin with the qualifiers A::B or C.

Note: If you want to use the compiler option **-qfunctrace** to disable tracing for a given function or a list of functions, you must use its suboption **-** followed by the names of the functions. For details about how to use **-qfunctrace** and its related suboptions, see "-qfunctrace" on page 146.

Examples
```
#pragma nofunctrace(a,b,c)
```

Related information
- "-qfunctrace" on page 146

#pragma nosimd
See "-qhot" on page 155.

#pragma novector
See "-qhot" on page 155.

#pragma options
Category

Language element control

Purpose

Specifies compiler options in your source program.

Syntax

```
►►──#──pragma──┬─option──┬──┬─── option_keyword ──────────────────────────┬──►◄
               └─options─┘  │              ┌────;────┐                     │
                            │              ├────────┐│                     │
                            │              │   ┌─,─┐ ││                     │
                            └─ option_keyword ──=──▼─ value ─┘             │
```
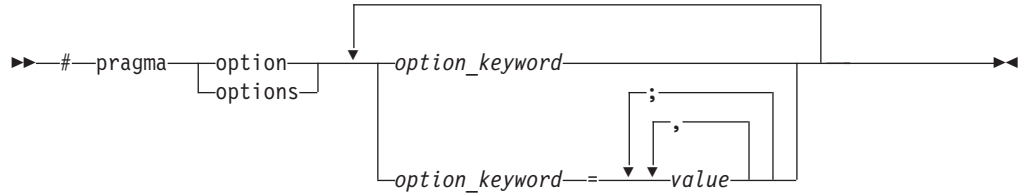
Parameters

The settings in the table below are valid *options* for **#pragma options**. For more information, see the pages of the equivalent compiler option.

Valid settings for #pragma options *option_keyword*	Compiler option equivalent
align=*option*	"-qalign" on page 91
[no]attr	"-qattr" on page 102
attr=full	
chars=*option*	"-qchars" on page 109
[no]check	"-qcheck" on page 110
[no]compact	"-qcompact" on page 114
[no]dbcs	"-qmbcs, -qdbcs" on page 221
[no]digraph	"-qdigraph" on page 124
[no]dollar	"-qdollar" on page 125
enum=*option*	"-qenum" on page 129
flag=*option*	"-qflag" on page 136
float=[no]*option*	"-qfloat" on page 137
[no]flttrap=*option*	"-qflttrap" on page 141
[no]fullpath	"-qfullpath" on page 145
halt	"-qhalt" on page 153
[no]idirfirst	"-qidirfirst" on page 159
[no]ignerrno	"-qignerrno" on page 160
ignprag=*option*	"-qignprag" on page 161
[no]info=*option*	"-qinfo" on page 164
initauto=*value*	"-qinitauto" on page 171
isolated_call=*names*	"-qisolated_call" on page 182
▶ C langlvl	"-qlanglvl" on page 188
[no]ldbl128	"-qldbl128" on page 206
[no]libansi	"-qlibansi" on page 208
[no]list	"-qlist" on page 210

Valid settings for #pragma options *option_keyword*	Compiler option equivalent
[no]longlong	"-qlonglong" on page 216
[no]maxmem=*number*	"-qmaxmem" on page 220
[no]mbcs	"-qmbcs, -qdbcs" on page 221
[no]optimize=*number*	"-O, -qoptimize" on page 227
▶ C++ priority=*number*	"-qpriority (C++ only)" on page 249
proclocal, procimported, procunknown	"-qprocimported, -qproclocal, -qprocunknown" on page 250
▶ C [no]proto	"-qproto (C only)" on page 252
[no]ro	"-qro" on page 258
[no]roconst	"-qroconst" on page 260
[no]showinc	"-qshowinc" on page 266
[no]source	"-qsource" on page 276
spill=*number*	"-qspill" on page 279
[no]stdinc	"-qstdinc" on page 285
[no]strict	"-qstrict" on page 286
tbtable=*option*	"-qtbtable" on page 296
tune=*option*	"-qtune" on page 308
[no]unrollunroll=*number*	"-qunroll" on page 312
▶ C [no]upconv	"-qupconv (C only)" on page 315
[no]xref	"-qxref" on page 325

Usage

Most **#pragma options** directives must come before any statements in your source program; only comments, blank lines, and other pragma specifications can precede them. For example, the first few lines of your program can be a comment followed by the **#pragma options** directive:

```
/* The following is an example of a #pragma options directive: */

#pragma options langlvl=stdc89 halt=s spill=1024 source

/* The rest of the source follows ... */
```

To specify more than one compiler option with the **#pragma options** directive, separate the options using a blank space. For example:

```
#pragma options langlvl=stdc89 halt=s spill=1024 source
```

#pragma option_override
Category

Optimization and tuning

Purpose

Allows you to specify optimization options at the subprogram level that override optimization options given on the command line.

This enables finer control of program optimization, and can help debug errors that occur only under optimization.

Syntax

```
►►──#──pragma──option_override──────────────────────────────────────────────►

►─(──identifier──,──"──opt──(──size──────────────────────)──"──)──────►◄
                                  ├─, yes─┤
                                  └─, no──┘
                     ├─level──,──0─┤
                           ├─2─┤
                           ├─3─┤
                           ├─4─┤
                           └─5─┘
                     ├─registerspillsize──,────size─┤
                     └─strict────────────────────────
                              ├─no────────┤
                              ├─all───────┤
                              ├─none──────┤
                              └─suboption_list─┘
```
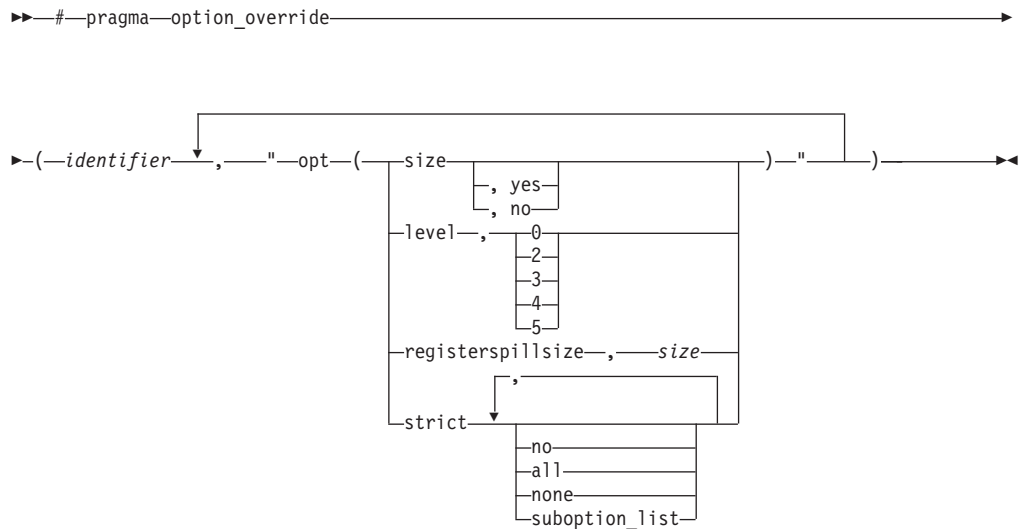
Parameters

identifier
    The name of a function for which optimization options are to be overridden.

The following table shows the equivalent command line option for each pragma suboption.

#pragma option_override value	Equivalent compiler option
level, 0	-O
level, 2	-O2
level, 3	-O3
level, 4	-O4
level, 5	-O5
registerspillsize, *size*	-qspill=*size*
size	-qcompact
size, yes	
size, no	-qnocompact
strict, all	-qstrict, -qstrict=all
strict, no, none	-qnostrict
strict, *suboption_list*	-qstrict=*suboption_list*

Defaults

See the descriptions for the options listed in the table above for default settings.

Usage

The pragma takes effect only if optimization is already enabled by a command-line option. You can only specify an optimization level in the pragma *lower* than the level applied to the rest of the program being compiled.

The **#pragma option_override** directive only affects functions that are defined in the same compilation unit. The pragma directive can appear anywhere in the translation unit. That is, it can appear before or after the function definition, before or after the function declaration, before or after the function has been referenced, and inside or outside the function definition.

▶ C++ This pragma cannot be used with overloaded member functions.

Examples

Suppose you compile the following code fragment containing the functions foo and faa using **-O2**. Since it contains the #pragma option_override(faa, "opt(level, 0)"), function faa will not be optimized.

```
foo(){
    .
    .
    .
    }

#pragma option_override(faa, "opt(level, 0)")

faa(){
    .
    .
    .
    }
```

Related information
- "-O, -qoptimize" on page 227
- "-qcompact" on page 114
- "-qspill" on page 279
- "-qstrict" on page 286

#pragma pack
Category

Object code control

Purpose

Sets the alignment of all aggregate members to a specified byte boundary.

If the byte boundary number is smaller than the natural alignment of a member, padding bytes are removed, thereby reducing the overall structure or union size.

The syntax and semantics of this pragma are different depending on the setting of the **-qpack_semantic** option.

Syntax

Default #pragma pack syntax   (-qpack_semantic=ibm in effect)

```
►►──#──pragma──pack──(─────────────)──────────────────────►◄
                         ├─nopack─┤
                         ├─number─┤
                         └─pop────┘
```
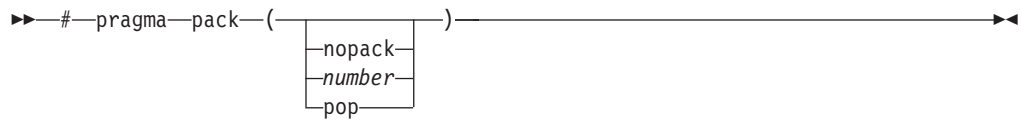
#pragma pack syntax with -qpack_semantic=gnu in effect

```
►►──#──pragma──pack──(─────────────────────────)──────────►◄
                        ├─number──push──────────┤
                        │           └─,──number─┘│
                        └─pop───────────────────┘
```
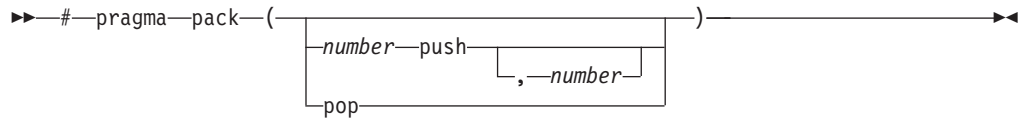
Defaults

Members of aggregates (structures, unions, and classes) are aligned on their natural boundaries and a structure ends on its natural boundary. The alignment of an aggregate is that of its strictest member (the member with the largest alignment requirement).

Parameters

nopack

Disables packing. Note that this parameter is not recognized when **-qpack_semantic=gnu** is in effect; a warning message is issued and the pragma is ignored.

push

When specified without a *number*, pushes whatever value is currently in effect to the top of the packing "stack". When used with a *number*, pushes that value to the top of the packing stack, and sets the packing value to that of *number* for structures that follow. Note that this parameter is not recognized when **-qpack_semantic=ibm** is in effect; a warning message is issued and the pragma is ignored.

number

is one of the following:

1   Aligns structure members on 1-byte boundaries, or on their natural alignment boundary, whichever is less.

2   Aligns structure members on 2-byte boundaries, or on their natural alignment boundary, whichever is less.

4   Aligns structure members on 4-byte boundaries, or on their natural alignment boundary, whichever is less.

8   Aligns structure members on 8-byte boundaries, or on their natural alignment boundary, whichever is less.

16 Aligns structure members on 16-byte boundaries, or on their natural alignment boundary, whichever is less.

pop

When **-qpack_semantic=ibm** is in effect, sets the packing rule to that which was in effect before the current setting. When **-qpack_semantic=gnu** is in

effect, pops the value specified in the last **push** statement off the stack and resets the current packing value to the value on the top of the stack, overriding any intervening value that may have been specified without a **push** statement.

Specifying **#pragma pack()** with no parameters (that is, with empty parentheses) has the following effect:

- Disables all packing (equivalent to specifying **#pragma pack(nopack)**), when **-qpack_semantic=ibm** is in effect.
- Sets the current packing value to that which was in effect at the beginning of the compilation unit, when **-qpack_semantic=gnu** is in effect.

Usage

The **#pragma pack** directive applies to the definition of an aggregate type, rather than to the declaration of an instance of that type; it therefore automatically applies to all variables declared of the specified type.

The **#pragma pack** directive modifies the current alignment rule for only the members of structures whose declarations follow the directive. It does not affect the alignment of the structure directly, but by affecting the alignment of the members of the structure, it may affect the alignment of the overall structure.

The **#pragma pack** directive cannot increase the alignment of a member, but rather can decrease the alignment. For example, for a member with data type of short, a **#pragma pack(1)** directive would cause that member to be packed in the structure on a 1-byte boundary, while a **#pragma pack(4)** directive would have no effect.

The **#pragma pack** directive causes bit fields to cross bit field container boundaries.

```
#pragma pack(2)
struct A{
int a:31;
int b:2;
}x;

int main(){
printf("size of S = %d\n", sizeof(s));
}

When compiled and run, the output is:
size of S = 6

But if you remove the #pragma pack directive, you get this output:
size of S = 8
```

The **#pragma pack** directive applies only to complete declarations of structures or unions; this excludes forward declarations, in which member lists are not specified. For example, in the following code fragment, the alignment for struct S is 4, since this is the rule in effect when the member list is declared:

```
#pragma pack(1)
struct S;
#pragma pack(4)
struct S { int i, j, k; };
```

A nested structure has the alignment that precedes its declaration, not the alignment of the structure in which it is contained, as shown in the following example:

```
#pragma pack (4)                    // 4-byte alignment
        struct nested {
          int  x;
          char y;
          int  z;
        };

        #pragma pack(1)          // 1-byte alignment
        struct packedcxx{          char   a;
          short  b;
          struct nested  s1;     // 4-byte alignment
        };
```

If more than one **#pragma pack** directive appears in a structure defined in an inlined function, the **#pragma pack** directive in effect at the beginning of the structure takes precedence.

Examples

The following example shows how the **#pragma pack** directive can be used to set the alignment of a structure definition:

```
//  header file file.h

   #pragma pack(1)

   struct jeff{           //    this structure is packed
     short bill;          //    along 1-byte boundaries
     int *chris;
   };
   #pragma pack(pop)       //   reset to previous alignment rule
// source file anyfile.c

   #include "file.h"

   struct jeff j;           //   uses the alignment specified
                            //   by the pragma pack directive
                            //   in the header file and is
                            //   packed along 1-byte boundaries
```

This example shows how a **#pragma pack** directive can affect the size and mapping of a structure:

```
struct s_t {
 char a;
 int b;
 short c;
 int d;
}S;
```

Default mapping:	With #pragma pack(1):
size of s_t = 16	size of s_t = 11
offset of a = 0	offset of a = 0
offset of b = 4	offset of b = 1
offset of c = 8	offset of c = 5
offset of d = 12	offset of d = 7
alignment of a = 1	alignment of a = 1
alignment of b = 4	alignment of b = 1
alignment of c = 2	alignment of c = 1
alignment of d = 4	alignment of d = 1

The following example defines a union uu containing a structure as one of its members, and declares an array of 2 unions of type uu:

```
union uu {
  short    a;
  struct {
    char x;
    char y;
    char z;
  } b;
};

union uu nonpacked[2];
```

Since the largest alignment requirement among the union members is that of short a, namely, 2 bytes, one byte of padding is added at the end of each union in the array to enforce this requirement:

```
 ┌──── nonpacked[0] ────┬──── nonpacked[1] ────┐
 │       a       │      │       a       │      │
 │  x    │   y   │  z   │  x    │   y   │  z   │
 └───────┴───────┴──────┴───────┴───────┴──────┘
 0    1    2    3    4    5    6    7    8
```
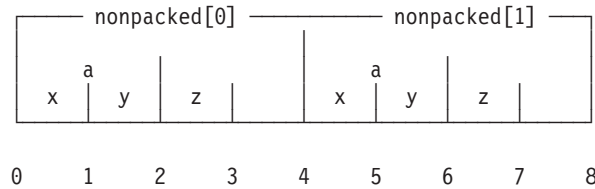
The next example uses **#pragma pack(1)** to set the alignment of unions of type uu to 1 byte:

```
#pragma pack(1)

union uu {
  short    a;
  struct {
    char x;
    char y;
    char z;
  } b;
};

union uu pack_array[2];
```

Now, each union in the array packed has a length of only 3 bytes, as opposed to the 4 bytes of the previous case:

```
 ┌──── packed[0] ────┬──── packed[1] ────┐
 │      a     │      │      a     │      │
 │  x   │  y  │  z   │  x   │  y  │  z   │
 └──────┴─────┴──────┴──────┴─────┴──────┘
 0    1    2    3    4    5    6
```
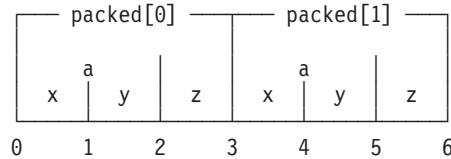
The following examples show the results of the differences in the semantics of this pragma depending on whether **-qpack_semantic=ibm** or **-qpack_semantic=ibm** is in effect.

This example shows the effect of specifying the **push** parameter:

```
#pragma pack(1)
#pragma pack(push)     // ignored when -qpack_semantic=ibm is in effect
#pragma pack(push,2)   // ignored when -qpack_semantic=ibm is in effect
```

```
struct s_t {
   char a;
   int b;
} S;
```

With **-qpack_semantic=gnu** in effect, the packing value in effect when the structure
S is declared is 2, and the structure is aligned on 2-byte boundaries. With
-qpack_semantic=ibm in effect, the second two directives are ignored, and the
packing value in effect for structure S is 1, and it is aligned on 1-byte boundaries.

This example shows the effect of specifying the **push** and **pop** parameters together:

```
#pragma pack(push,1)    // ignored when -qpack_semantic=ibm is in effect
#pragma pack(push,4)    // ignored when -qpack_semantic=ibm is in effect
#pragma pack(2)
#pragma pack(pop)
#pragma pack(pop)
#pragma pack(pop)

struct s_t {
   char a;
   int b;
} S;
```

With **-qpack_semantic=gnu** in effect, since **pop** only pops values that have been
pushed onto the stack with a **push** directive, the first **pop** directive pops 4 off the
stack, the second one pops 1 off the stack, and the alignment is the setting in effect
at the beginning of the compilation unit (the intervening **#pragma pack(2)** directive
is overridden). With **-qpack_semantic=ibm** in effect, the **pop** statement pops the
value 2 off the stack, and the alignment is the setting in effect at the beginning of
the compilation unit.

The following example shows the effect of specifying the directive inside a nested
structure:

```
struct s_t {
   char a;
   int b;

   #pragma pack(1)

   struct t_t {
      char x;
      int y;
   }T;

   char c;

   #pragma pack(2)
   #pragma pack(1)

   int d;

   #pragma align(natural)   \\ this only affects u_t.
   #pragma pack(2)          \\ this only affects u_t.

   struct u_t {
      char j;
      int k;
   }U;

}S;
```

When **-qpack_semantic=gnu** is in effect, the first **#pragma pack(1)** directive applies to both structure t_t *and* s_t. With **-qpack_semantic=ibm** the first **#pragma pack(1)** directive applies to structure t_t only.

Related information

- "-qalign" on page 91
- "-qpack_semantic" on page 233
- "Using alignment modifiers" in the *XL C/C++ Optimization and Programming Guide*

#pragma priority (C++ only)

See "-qpriority (C++ only)" on page 249.

#pragma reachable
Category

Optimization and tuning

Purpose

Informs the compiler that the point in the program after a named function can be the target of a branch from some unknown location.

By informing the compiler that the instruction after the specified function can be reached from a point in your program other than the return statement in the named function, the pragma allows for additional opportunities for optimization.

Note: The compiler automatically inserts **#pragma reachable** directives for the setjmp family of functions (setjmp, _setjmp, sigsetjmp, and _sigsetjmp) when you include the setjmp.h header file.

Syntax

```
►►─#─pragma─reachable─(──▼─function_name──┘─)──────────────►◄
                          └──────,──────┘
```

Parameters

function_name
    The name of a function preceding the instruction which is reachable from a point in the program other than the function's return statement.

Defaults

Not applicable.

Related information

- "#pragma leaves" on page 350

#pragma reg_killed_by
Category

Optimization and tuning

Purpose

Specifies registers that may be altered by functions specified by **#pragma mc_func**.

Ordinarily, code generated for functions specified by **#pragma mc_func** may alter any or all volatile registers available on your system. You can use **#pragma reg_killed_by** to explicitly list a specific set of volatile registers to be altered by such functions. Registers not in this list will not be altered.

Syntax

```
►►──#──pragma──reg_killed_by──function──┬─────────────────────────┬──►◄
                                        │    ┌──,──────────┐       │
                                        └─▼──register──────┴───────┘
                                             └──-──register──┘
```
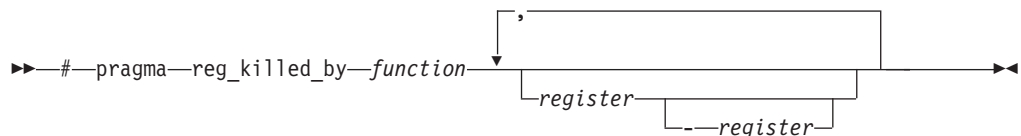
Parameters

function
> The name of a function previously defined using the **#pragma mc_func** directive.

register
> The symbolic name(s) of either a single register or a range of registers to be altered by the named *function*. The symbolic name must be a valid register name on the target platform. Valid registers are:

> **cr0, cr1, and cr5 to cr7**
>> Condition registers

> **ctr**　Count register

> **gr0 and gr3 to gr12**
>> General purpose registers

> **fp0 to fp13**
>> Floating-point registers

> **fs**　Floating point and status control register

> **lr**　Link register

> **vr0 to vr31**
>> Vector registers (on selected processors only)

> **xer**　Fixed-point exception register

> You can identify a range of registers by providing the symbolic names of both starting and ending registers, separated by a dash.

> If no *register* is specified, no volatile registers will be killed by the named *function*.

Examples

The following example shows how to use **#pragma reg_killed_by** to list a specific set of volatile registers to be used by the function defined by **#pragma mc_func**.

```
int add_logical(int, int);
#pragma mc_func add_logical {"7c632014" "7c630194"}
                /*    addc        r3 <- r3, r4          */
                /*    addze       r3 <- r3, carry bit   */

#pragma reg_killed_by add_logical gr3, xer
                /* only gpr3 and the xer are altered by this function */


main() {

    int i,j,k;

    i = 4;
    k = -4;
    j = add_logical(i,k);
    printf("\n\nresult = %d\n\n",j);
}
```

Related information
- "#pragma mc_func" on page 354

#pragma report (C++ only)
Category

Listings, messages and compiler information

Purpose

Controls the generation of diagnostic messages.

The pragma allows you to specify a minimum severity level for a message for it to display, or allows you to enable or disable a specific message regardless of the prevailing report level.

Syntax
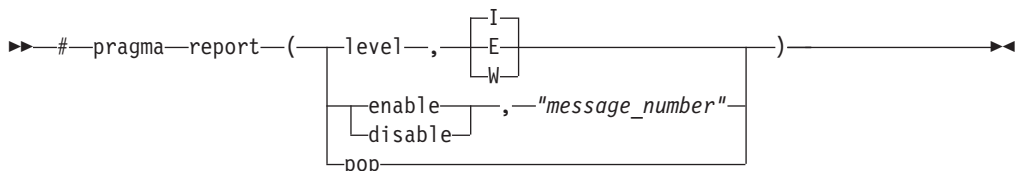
Defaults

The default report level is Informational (**I**), which displays messages of all types.

Parameters

level
   Indicates that the pragma is set according to the minimum severity level of diagnostic messages to display.

E Indicates that only error messages will display. Error messages are of the highest severity. This is equivalent to the **-qflag=e:e** compiler option.

W Indicates that warning and error messages will display. This is equivalent to the **-qflag=w:w** compiler option.

I Indicates that all diagnostic messages will display: warning, error and informational messages. Informational messages are of the lowest severity. This is equivalent to the **-qflag=i:i** compiler option.

enable
Enables the specified *"message_number"*.

disable
Disables the specified *"message_number"*.

"message_number"
Represents a message identifier, which consists of a prefix followed by the message number in quotation marks; for example, "CCN1004".

Note: You must use quotation marks with *message_number* as in the preceding example "CCN1004".

pop
Reverts the report level to that which was previously in effect. If no previous report level has been specified, a warning is issued, and the report level remains unchanged.

Usage

The pragma takes precedence over **#pragma info** and most compiler options. For example, if you use **#pragma report** to disable a compiler message, that message will not be displayed with any **-qflag** compiler option setting.

Related information
- "-qflag" on page 136

#pragma STDC cx_limited_range
Category

Optimization and tuning

Purpose

Instructs the compiler that complex division and absolute value are only invoked with values such that intermediate calculation will not overflow or lose significance.

Syntax

```
>>--#--pragma--STDC cx_limited_range--+--off--+----------------><
                                      +--on---+
                                      +-default-+
```
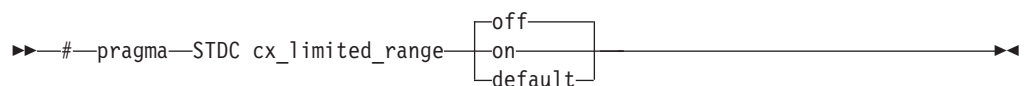
Usage

Using values outside the limited range may generate wrong results, where the limited range is defined such that the "obvious symbolic definition" will not overflow or run out of precision.

The pragma is effective from its first occurrence until another **cx_limited_range** pragma is encountered, or until the end of the translation unit. When the pragma occurs inside a compound statement (including within a nested compound statement), it is effective from its first occurrence until another **cx_limited_range** pragma is encountered, or until the end of the compound statement.

Examples

The following example shows the use of the pragma for complex division:

```
#include <complex.h>

_Complex double a, b, c, d;
void p() {

d = b/c;

{

#pragma STDC CX_LIMITED_RANGE ON

a = b / c;

}
}
```

The following example shows the use of the pragma for complex absolute value:

```
#include <complex.h>

_Complex double cd = 10.10 + 10.10*I;
int p() {

#pragma STDC CX_LIMITED_RANGE ON

double d = cabs(cd);
}
```

Related information
- "Standard pragmas" in the *XL C/C++ Language Reference*

#pragma stream_unroll
Category

Optimization and tuning

Purpose

When optimization is enabled, breaks a stream contained in a `for` loop into multiple streams.

Syntax

```
►►──#──pragma──stream_unroll─┬──────────────┬──────────────────────────►◄
                             └─(─number─)─┘
```

Parameters

number

    A loop unrolling factor. ► **C** The value of *number* is a positive integral constant expression. ► **C++** The value of *number* is a positive scalar integer or compile-time constant initialization expression.

    An unroll factor of 1 disables unrolling.

If *number* is not specified, the optimizer determines an appropriate unrolling factor for each nested loop.

Usage

To enable stream unrolling, you must specify **-qhot** and **-qstrict**, or **-qsmp**, or use optimization level **-O4** or higher. If **-qstrict** is in effect, no stream unrolling takes place.

For stream unrolling to occur, the **#pragma stream_unroll** directive must be the last pragma specified preceding a for loop. ► **C** Specifying **#pragma stream_unroll** more than once for the same for loop or combining it with other loop unrolling pragmas (**#pragma unroll**, **#pragma nounroll**, **#pragma unrollandfuse**, **#pragma nounrollandfuse**) results in a warning. ► **C++** The compiler silently ignores all but the last of multiple loop unrolling pragmas specified on the same for loop.

Examples

The following is an example of how **#pragma stream_unroll** can increase performance.

```
int i, m, n;
int a[1000][1000];
int b[1000][1000];
int c[1000][1000];


....

#pragma stream_unroll(4)
for (i=1; i<n; i++) {
    a[i] = b[i] * c[i];
}
```

The unroll factor of 4 reduces the number of iterations from n to n/4, as follows:

```
for (i=1; i<n/4; i++) {
    a[i] = b[i] + c[i];
    a[i+m] = b[i+m] + c[i+m];
    a[i+2*m] = b[i+2*m] + c[i+2*m];
    a[i+3*m] = b[i+3*m] + c[i+3*m];
}
```

Related information
- "-qunroll" on page 312
- "#pragma unrollandfuse" on page 372

#pragma strings

See "-qro" on page 258.

#pragma unroll

See "-qunroll" on page 312.

#pragma unrollandfuse
Category

Optimization and tuning

Purpose

Instructs the compiler to attempt an unroll and fuse operation on nested `for` loops.

Syntax

```
►►──#──pragma──┬─nounrollandfuse─────────────────────┬──────────────────►◄
               └─unrollandfuse─┬─────────────────┬───┘
                               └─(──number──)─────┘
```

Parameters

number

> A loop unrolling factor. ▶ **C** The value of *number* is a positive integral constant expression. ▶ **C++** The value of *number* is a positive scalar integer or compile-time constant initialization expression.

If *number* is not specified, the optimizer determines an appropriate unrolling factor for each nested loop.

Usage

The **#pragma unrollandfuse** directive applies only to the outer loops of nested `for` loops that meet the following conditions:

- There must be only one loop counter variable, one increment point for that variable, and one termination variable. These cannot be altered at any point in the loop nest.
- Loops cannot have multiple entry and exit points. The loop termination must be the only means to exit the loop.
- Dependencies in the loop must not be "backwards-looking". For example, a statement such as A[i][j] = A**[i -1]**[j + 1] + 4) must not appear within the loop.

For loop unrolling to occur, the **#pragma unrollandfuse** directive must precede a `for` loop. You must not specify **#pragma unrollandfuse** for the innermost `for` loop.

You must not specify **#pragma unrollandfuse** more than once, or combine the directive with **#pragma nounrollandfuse**, **#pragma nounroll**, **#pragma unroll**, or **#pragma stream_unroll** directives for the same `for` loop.

Predefined macros

None.

Examples

In the following example, a **#pragma unrollandfuse** directive replicates and fuses the body of the loop. This reduces the number of cache misses for array b.

```
int i, j;
int a[1000][1000];
int b[1000][1000];
int c[1000][1000];


....

#pragma unrollandfuse(2)
for (i=1; i<1000; i++) {
    for (j=1; j<1000; j++) {
        a[j][i] = b[i][j] * c[j][i];
    }
}
```

The `for` loop below shows a possible result of applying the **#pragma unrollandfuse(2)** directive to the loop shown above:

```
for (i=1; i<1000; i=i+2) {
    for (j=1; j<1000; j++) {
        a[j][i] = b[i][j] * c[j][i];
        a[j][i+1] = b[i+1][j] * c[j][i+1];
    }
}
```

You can also specify multiple **#pragma unrollandfuse** directives in a nested loop structure.

```
int i, j, k;
int a[1000][1000];
int b[1000][1000];
int c[1000][1000];
int d[1000][1000];
int e[1000][1000];


....

#pragma unrollandfuse(4)
for (i=1; i<1000; i++) {
#pragma unrollandfuse(2)
    for (j=1; j<1000; j++) {
    for (k=1; k<1000; k++) {
        a[j][i] = b[i][j] * c[j][i] + d[j][k] * e[i][k];
        }
    }
}
```

Related information
- "-qunroll" on page 312
- "#pragma stream_unroll" on page 370

#pragma weak

Category

Object code control

Purpose

Prevents the linker from issuing error messages if it encounters a symbol multiply-defined during linking, or if it does not find a definition for a symbol.

The pragma can be used to allow a program to call a user-defined function that has the same name as a library function. By marking the library function definition as "weak", the programmer can reference a "strong" version of the function and cause the linker to accept multiple definitions of a global symbol in the object code. While this pragma is intended for use primarily with functions, it will also work for most data objects.

Syntax

```
►►─#─pragma─weak─name1──────────────────────────────────►◄
                        └─=─name2─┘
```

Parameters

name1
    A name of a data object or function with external linkage.

name2
    A name of a data object or function with external linkage.

    ▶ **C++**   *name2* must not be a member function. If *name2* is a template function, you must explicitly instantiate the template function.

▶ **C++**   Names must be specified using their mangled names. To obtain C++ mangled names, compile your source to object files only, using the **-c** compiler option, and use the **nm** operating system command on the resulting object file. (See also "Name mangling" in the *XL C/C++ Language Reference* for details on using the extern "C" linkage specifier on declarations to prevent name mangling.)

Usage

There are two forms of the **weak** pragma:

#pragma weak *name1*
        This form of the pragma marks the definition of the *name1* as "weak" in a given compilation unit. If *name1* is referenced from anywhere in the program, the linker will use the "strong" version of the definition (that is, the definition not marked with **#pragma weak**), if there is one. If there is no strong definition, the linker will use the weak definition; if there are multiple weak definitions, it is unspecified which weak definition the linker will select (typically, it uses the definition found in the first object file specified on the command line during the link step). *name1* must be defined in the same compilation unit as **#pragma weak**. If *name1* is referenced, but no definition of it can be found, it is assigned a value of 0.

#pragma weak *name1=name2*

> This form of the pragma creates a weak definition of the *name1* for a given compilation unit, and an alias for *name2*. If *name1* is referenced from anywhere in the program, the linker will use the "strong" version of the definition (that is, the definition not marked with **#pragma weak**), if there is one. If there is no strong definition, the linker will use the weak definition, which resolves to the definition of *name2*. If there are multiple weak definitions, it is unspecified which weak definition the linker will select (typically, it uses the definition found in the first object file specified on the command line during the link step).
>
> *name2* must be defined in the same compilation unit as **#pragma weak**. *name1* may or may not be declared in the same compilation unit as the **#pragma weak**, but must never be defined in the compilation unit. If *name1* is declared in the compilation unit, *name1*'s declaration must be compatible to that of *name2*. For example, if *name2* is a function, *name1* must have the same return and argument types as *name2*.

This pragma should not be used with uninitialized global data, or with shared library data objects that are exported to executables.

Examples

The following is an example of the **#pragma weak** *name1* form:

```
// Compilation unit 1:

#include <stdio.h>

void foo();

int main()
{
        foo();
}

// Compilation unit 2:

#include <stdio.h>

#if __cplusplus
#pragma weak _Z3foov
#else
#pragma weak foo
#endif
void foo()
{
        printf("Foo called from compilation unit 2\n");
}

// Compilation unit 3:

#include <stdio.h>

void foo()
{
        printf("Foo called from compilation unit 3\n");
}
```

If all three compilation units are compiled and linked together, the linker will use the strong definition of foo in compilation unit 3 for the call to foo in compilation unit 1, and the output will be:

```
Foo called from compilation unit 3
```

If only compilation unit 1 and 2 are compiled and linked together, the linker will use the weak definition of foo in compilation unit 2, and the output will be:

```
Foo called from compilation unit 2
```

The following is an example of the **#pragma weak** *name1*=*name2* form:

```
// Compilation unit 1:

#include <stdio.h>

void foo();

int main()
{
foo();
}

// Compilation unit 2:

#include <stdio.h>

void foo(); // optional

#if __cplusplus
#pragma weak _Z3foov = _Z4foo2v
#else#pragma weak foo = foo2
#endif
void foo2()
{
printf("Hello from foo2!\n");
}

// Compilation unit 3:

#include <stdio.h>

void foo()
{
printf("Hello from foo!\n");
}
```

If all three compilation units are compiled and linked together, the linker will use the strong definition of foo in compilation unit 3 for the call to foo from compilation unit 1, and the output will be:

```
Hello from foo!
```

If only compilation unit 1 and 2 are compiled and linked together, the linker will use the weak definition of foo in compilation unit 2, which is an alias for foo2, and the output will be:

```
Hello from foo2!
```

Related information

- "The weak variable attribute" in the *XL C/C++ Language Reference*
- "The weak function attribute" in the *XL C/C++ Language Reference*
- "#pragma map" on page 351

Pragma directives for parallel processing

Parallel processing operations are controlled by pragma directives in your program source. The pragmas have effect only when parallelization is enabled with the **-qsmp** compiler option.

#pragma omp atomic
Purpose

The **omp atomic** directive identifies a specific memory location that must be updated atomically and not be exposed to multiple, simultaneous writing threads.

Syntax

```
►►─#─pragma─omp atomic────────────────────────────────────────────────►◄
                        └─statement─┘
```

where *statement* is an expression statement of scalar type that takes one of the forms that follow:

statement	Conditions
x bin_op = *expr*	where: *bin_op* is one of: + * - / & ^ | << >> *expr* is an expression of scalar type that does not reference *x*.
x++	
++x	
x--	
--x	

Usage

Load and store operations are atomic only for object *x*. Evaluation of *expr* is not atomic.

All atomic references to a given object in your program must have a compatible type.

Objects that can be updated in parallel and may be subject to race conditions should be protected with the **omp atomic** directive.

Examples

```
extern float x[], *p = x, y;

/* Protect against race conditions among multiple updates.  */
#pragma omp atomic
x[index[i]] += y;

/* Protect against races with updates through x.            */
#pragma omp atomic
p[i] -= 1.0f;
```

#pragma omp parallel
Purpose

The **omp parallel** directive explicitly instructs the compiler to parallelize the chosen block of code.

Syntax

```
►►—#—pragma—omp parallel——clause—————————————————►◄
```

Parameters

clause is any of the following:

if (*exp***)**
When the `if` argument is specified, the program code executes in parallel only if the scalar expression represented by *exp* evaluates to a nonzero value at run time. Only one `if` clause can be specified.

private (*list***)**
Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

firstprivate (*list***)**
Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized with the value of the original variable as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

num_threads (*int_exp***)**
The value of *int_exp* is an integer expression that specifies the number of threads to use for the parallel region. If dynamic adjustment of the number of threads is also enabled, then *int_exp* specifies the maximum number of threads to be used.

shared (*list***)**
Declares the scope of the comma-separated data variables in *list* to be shared across all threads.

default (shared | none)
Defines the default data scope of variables in each thread. Only one **default** clause can be specified on an **omp parallel** directive.

Specifying **default(shared)** is equivalent to stating each variable in a **shared(***list***)** clause.

Specifying **default(none)** requires that each data variable visible to the parallelized statement block must be explcitly listed in a data scope clause, with the exception of those variables that are:
- const-qualified,
- specified in an enclosed data scope attribute clause, or,
- used as a loop control variable referenced only by a corresponding **omp for** or **omp parallel for** directive.

copyin (*list***)**
For each data variable specified in *list*, the value of the data variable in the

master thread is copied to the thread-private copies at the beginning of the parallel region. Data variables in *list* are separated by commas.

Each data variable specified in the **copyin** clause must be a **threadprivate** variable.

reduction (*operator***:** *list***)**

Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas.

A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back into the original value of the shared reduction variable.

Variables specified in the **reduction** clause:
- Must be of a type appropriate to the operator.
- Must be shared in the enclosing context.
- Must not be const-qualified.
- Must not have pointer type.

Usage

When a parallel region is encountered, a logical team of threads is formed. Each thread in the team executes all statements within a parallel region except for work-sharing constructs. Work within work-sharing constructs is distributed among the threads in a team.

Loop iterations must be independent before the loop can be parallelized. An implied barrier exists at the end of a parallelized statement block.

Nested parallel regions are always serialized.

#pragma omp for
Purpose

The **omp for** directive instructs the compiler to distribute loop iterations within the team of threads that encounters this work-sharing construct.

Syntax

```
                                        ,
                              ┌─────────◄─────────┐
►►──#──pragma──omp for──┬─────┴───────────────────┴─────┬──────────►◄
                        │                               │
                        └─clause─for-loop─┘
```
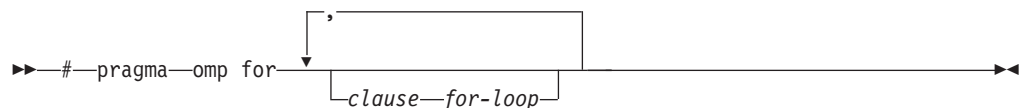
Parameters

clause is any of the following:

collapse (*n***)**

Specifying the **collapse** clause allows you to parallelize multiple loops in a nest without introducing nested parallelism. See the "collapse" on page 382 topic for more information.

private (*list***)**

Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

firstprivate (*list***)**

Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

lastprivate (*list***)**

Declares the scope of the data variables in *list* to be private to each thread. The final value of each variable in *list*, if assigned, will be the value assigned to that variable in the last iteration. Variables not assigned a value will have an indeterminate value. Data variables in *list* are separated by commas.

reduction (*operator***:***list***)**

Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas.

A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back into the original value of the shared reduction variable.

Variables specified in the **reduction** clause:
- must be of a type appropriate to the operator.
- must be shared in the enclosing context.
- must not be const-qualified.
- must not have pointer type.

ordered

Specify this clause if an ordered construct is present within the dynamic extent of the **omp for** directive.

schedule (*type***)**

Specifies how iterations of the **for** loop are divided among available threads. Acceptable values for *type* are:

auto   With **auto**, scheduling is delegated to the compiler and runtime system. The compiler and runtime system can choose any possible mapping of iterations to threads (including all possible valid schedules) and these may be different in different loops.

dynamic

Iterations of a loop are divided into chunks of size **ceiling**(*number_of_iterations*/*number_of_threads*).

Chunks are dynamically assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.

dynamic,*n*

As above, except chunks are set to size *n*. *n* must be an integral assignment expression of value 1 or greater.

guided

Chunks are made progressively smaller until the default minimum chunk size is reached. The first chunk is of size **ceiling**(*number_of_iterations*/*number_of_threads*). Remaining chunks are of size **ceiling**(*number_of_iterations_left*/*number_of_threads*).

The minimum chunk size is 1.

Chunks are assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.

guided,*n*

As above, except the minimum chunk size is set to *n*. *n* must be an integral assignment expression of value 1 or greater.

runtime

Scheduling policy is determined at run time. Use the OMP_SCHEDULE environment variable to set the scheduling type and chunk size.

static   Iterations of a loop are divided into chunks of size **ceiling**(*number_of_iterations/number_of_threads*). Each thread is assigned a separate chunk.

This scheduling policy is also known as *block scheduling*.

static,*n*

Iterations of a loop are divided into chunks of size *n*. Each chunk is assigned to a thread in *round-robin* fashion.

n must be an integral assignment expression of value 1 or greater.

This scheduling policy is also known as *block cyclic scheduling*.

Note: if *n*=1, iterations of a loop are divided into chunks of size 1 and each chunk is assigned to a thread in *round-robin* fashion. This scheduling policy is also known as *block cyclic scheduling*

nowait

Use this clause to avoid the implied **barrier** at the end of the **for** directive. This is useful if you have multiple independent work-sharing sections or iterative loops within a given parallel region. Only one **nowait** clause can appear on a given **for** directive.

and where *for_loop* is a for loop construct with the following canonical shape:

```
for (init_expr; exit_cond; incr_expr)
 statement
```

where:

init_expr	takes form:	`iv = b` `integer-type iv = b`

exit_cond	takes form:	`iv <= ub`
		`iv <  ub`
		`iv >= ub`
		`iv >  ub`

incr_expr	takes form:	`++iv`
		`iv++`
		`--iv`
		`iv--`
		`iv += incr`
		`iv -= incr`
		`iv = iv + incr`
		`iv = incr + iv`
		`iv = iv - incr`

and where:

iv	Iteration variable. The iteration variable must be a `signed integer` not modified anywhere within the for loop. It is implicitly made private for the duration of the for operation. If not specified as **lastprivate**, the iteration variable will have an indeterminate value after the operation completes.
b, ub, incr	Loop invariant signed integer expressions. No synchronization is performed when evaluating these expressions and evaluated side effects may result in indeterminate values.

Usage

This pragma must appear immediately before the loop or loop block directive to be affected.

Program sections using the **omp for** pragma must be able to produce a correct result regardless of which thread executes a particular iteration. Similarly, program correctness must not rely on using a particular scheduling algorithm.

The for loop iteration variable is implicitly made private in scope for the duration of loop execution. This variable must not be modified within the body of the for loop. The value of the increment variable is indeterminate unless the variable is specified as having a data scope of **lastprivate**.

An implicit barrier exists at the end of the for loop unless the **nowait** clause is specified.

Restrictions are:
- The for loop must be a structured block, and must not be terminated by a break statement.
- Values of the loop control expressions must be the same for all iterations of the loop.
- An **omp for** directive can accept only one **schedule** clauses.
- The value of *n* (chunk size) must be the same for all threads of a parallel region.

Related reference

"collapse"

collapse
Purpose

Specifying the **collapse** clause allows you to parallelize multiple loops in a nest without introducing nested parallelism. This clause is used with the **for** and **parallel for** pragmas.

Syntax

```
►►—COLLAPSE—(—n—)————————————————————————►◄
```
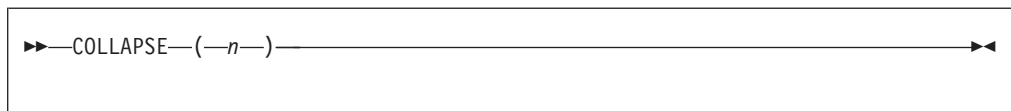
Rules
- Only one collapse clause is allowed on a worksharing **for** or **parallel for** pragma.

- The specified number of loops must be present lexically. That is, none of the loops can be in a called subroutine.
- The loops must form a rectangular iteration space and the bounds and stride of each loop must be invariant over all the loops.
- If the loop indices are of different size, the index with the largest size will be used for the collapsed loop.
- The loops must be perfectly nested; that is, there is no intervening code nor any OpenMP pragma between the loops which are collapsed.
- The associated do-loops must be structured blocks. Their execution must not be terminated by an **break** statement.
- If multiple loops are associated to the loop construct, only an iteration of the innermost associated loop may be curtailed by a **continue** statement. If multiple loops are associated to the loop construct, there must be no branches to any of the loop termination statements except for the innermost associated loop.

Ordered construct
> During execution of an iteration of a loop or a loop nest within a loop region, the executing thread must not execute more than one ordered region which binds to the same loop region. As a consequence, if multiple loops are associated to the loop construct by a collapse clause, the ordered construct has to be located inside all associated loops.

Lastprivate clause
> When a lastprivate clause appears on the pragma that identifies a work-sharing construct, the value of each new list item from the sequentially last iteration of the associated loops, is assigned to the original list item even if a collapse clause is associated with the loop

Other SMP and performance pragmas
> **stream_unroll**,**unroll**,**unrollandfuse**,**nonunrollandfuse** pragmas cannot be used for any of the loops associated with the **collapse** clause loop nest.

Related reference

"#pragma omp for" on page 379

"#pragma omp parallel for" on page 384

#pragma omp ordered
Purpose

The **omp ordered** directive identifies a structured block of code that must be executed in sequential order.

Syntax

```
►►──#──pragma──omp ordered──────────────────────────────────────────────►◄
```

Usage

The **omp ordered** directive must be used as follows:

- It must appear within the extent of a **omp for** or **omp parallel for** construct containing an **ordered** clause.
- It applies to the statement block immediately following it. Statements in that block are executed in the same order in which iterations are executed in a sequential loop.

- An iteration of a loop must not execute the same **omp ordered** directive more than once.
- An iteration of a loop must not execute more than one distinct **omp ordered** directive.

#pragma omp parallel for
Purpose

The **omp parallel for** directive effectively combines the **omp parallel** and **omp for** directives. This directive lets you define a parallel region containing a single **for** directive in one step.

Syntax

```
         ┌──────────┐
         │    ┌─┐   │
         │    └,┘   │
         ▼          │
►►──#──pragma──omp parallel for──┬──────────────────────┬──────────────────►◄
                                 └─clause──for-loop──────┘
```
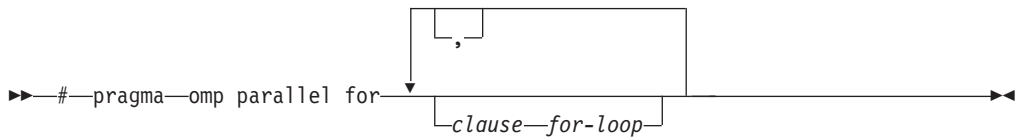
Usage

With the exception of the **nowait** clause, clauses and restrictions described in the **omp parallel** and **omp for** directives also apply to the **omp parallel for** directive.

#pragma omp section, #pragma omp sections
Purpose

The **omp sections** directive distributes work among threads bound to a defined parallel region.

Syntax

```
                         ┌─,─┐
                         ▼   │
►►──#──pragma──omp sections──clause──────────────────────────────────────►◄
```

Parameters

clause is any of the following:

private (*list*)
Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

firstprivate (*list*)
Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

lastprivate (*list*)
Declares the scope of the data variables in *list* to be private to each thread. The final value of each variable in *list*, if assigned, will be the value assigned to that variable in the last **section**. Variables not assigned a value will have an indeterminate value. Data variables in *list* are separated by commas.

reduction (*operator***:** *list***)**

> Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas.

> A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back into the original value of the shared reduction variable.

> Variables specified in the **reduction** clause:
> * must be of a type appropriate to the operator.
> * must be shared in the enclosing context.
> * must not be const-qualified.
> * must not have pointer type.

nowait

> Use this clause to avoid the implied **barrier** at the end of the **sections** directive. This is useful if you have multiple independent work-sharing sections within a given parallel region. Only one **nowait** clause can appear on a given **sections** directive.

Usage

The **omp section** directive is optional for the first program code segment inside the **omp sections** directive. Following segments must be preceded by an **omp section** directive. All **omp section** directives must appear within the lexical construct of the program source code segment associated with the **omp sections** directive.

When program execution reaches a **omp sections** directive, program segments defined by the following **omp section** directive are distributed for parallel execution among available threads. A barrier is implicitly defined at the end of the larger program region associated with the **omp sections** directive unless the **nowait** clause is specified.

#pragma omp parallel sections
Purpose

The **omp parallel sections** directive effectively combines the **omp parallel** and **omp sections** directives. This directive lets you define a parallel region containing a single **sections** directive in one step.

Syntax

```
►►—#—pragma—omp parallel sections────────┬──────────────────►◄
                                    ┌──,──┐
                                    ▼     │
                                 └─clause─┘
```
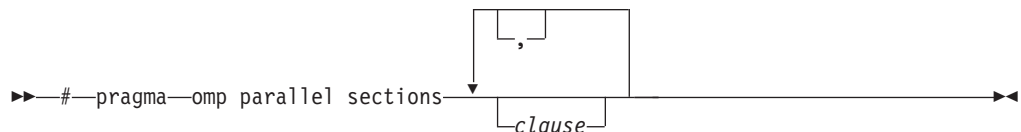
Usage

All clauses and restrictions described in the **omp parallel** and **omp sections** directives apply to the **omp parallel sections** directive.

#pragma omp single
Purpose

The **omp single** directive identifies a section of code that must be run by a single available thread.

Syntax
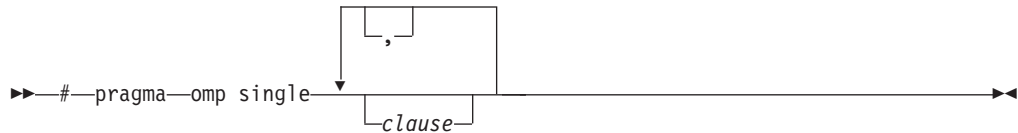
Parameters

clause is any of the following:

private (*list***)**
> Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.
>
> A variable in the **private** clause must not also appear in a **copyprivate** clause for the same **omp single** directive.

copyprivate (*list***)**
> Broadcasts the values of variables specified in *list* from one member of the team to other members. This occurs after the execution of the structured block associated with the **omp single** directive, and before any of the threads leave the barrier at the end of the construct. For all other threads in the team, each variable in the *list* becomes defined with the value of the corresponding variable in the thread that executed the structured block. Data variables in *list* are separated by commas. Usage restrictions for this clause are:
> * A variable in the **copyprivate** clause must not also appear in a **private** or **firstprivate** clause for the same **omp single** directive.
> * If an **omp single** directive with a **copyprivate** clause is encountered in the dynamic extent of a parallel region, all variables specified in the **copyprivate** clause must be private in the enclosing context.
> * Variables specified in **copyprivate** clause within dynamic extent of a parallel region must be private in the enclosing context.
> * A variable that is specified in the **copyprivate** clause must have an accessible and unambiguous copy assignment operator.
> * The **copyprivate** clause must not be used together with the **nowait** clause.

firstprivate (*list***)**
> Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.
>
> A variable in the **firstprivate** clause must not also appear in a **copyprivate** clause for the same **omp single** directive.

nowait
> Use this clause to avoid the implied **barrier** at the end of the **single** directive. Only one **nowait** clause can appear on a given **single** directive. The **nowait** clause must not be used together with the **copyprivate** clause.

Usage

An implied barrier exists at the end of a parallelized statement block unless the **nowait** clause is specified.

#pragma omp master
Purpose

The **omp master** directive identifies a section of code that must be run only by the master thread.

Syntax

►►—#—pragma—omp master————————————————————————————►◄

Usage

Threads other than the master thread will not execute the statement block associated with this construct.

No implied barrier exists on either entry to or exit from the master section.

#pragma omp critical
Purpose

The **omp critical** directive identifies a section of code that must be executed by a single thread at a time.

Syntax

►►—#—pragma—omp critical———(name)————————————————————►◄

where *name* can optionally be used to identify the critical region. Identifiers naming a critical region have external linkage and occupy a namespace distinct from that used by ordinary identifiers.

Usage

A thread waits at the start of a critical region identified by a given name until no other thread in the program is executing a critical region with that same name. Critical sections not specifically named by **omp critical** directive invocation are mapped to the same unspecified name.

#pragma omp barrier
Purpose

The **omp barrier** directive identifies a synchronization point at which threads in a parallel region will wait until all other threads in that section reach the same point. Statement execution past the **omp barrier** point then continues in parallel.

Syntax

```
►►──#──pragma──omp barrier────────────────────────────────────────►◄
```

Usage

The **omp barrier** directive must appear within a block or compound statement. For example:

```
if (x!=0) {
   #pragma omp barrier    /* valid usage    */
}
if (x!=0)
   #pragma omp barrier    /* invalid usage */
```

#pragma omp flush
Purpose

The **omp 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.

Syntax

```
                             ┌─ , ─┐
                             │     │
                             ▼     │
►►──#──pragma──omp flush─────┴───────┴────────────────────────────────────►◄
                             └─list─┘
```

where *list* is a comma-separated list of variables that will be synchronized.

Usage

If *list* includes a pointer, the pointer is flushed, not the object being referred to by the pointer. If *list* is not specified, all shared objects are synchronized except those inaccessible with automatic storage duration.

An implied **flush** directive appears in conjunction with the following directives:
- **omp barrier**
- Entry to and exit from **omp critical**.
- Exit from **omp parallel**.
- Exit from **omp for**.
- Exit from **omp sections**.
- Exit from **omp single**.

The **omp flush** directive must appear within a block or compound statement. For example:

```
if (x!=0) {
   #pragma omp flush    /* valid usage    */
}
if (x!=0)
   #pragma omp flush    /* invalid usage */
```

#pragma omp threadprivate
Purpose

The **omp threadprivate** directive makes the named file-scope, namespace-scope, or static block-scope variables private to a thread.

Syntax

```
►►──#──pragma──omp threadprivate──┬──(identifier)──┬──────────────────────────►◄
                                  └──────,◄────────┘
```

where *identifier* is a file-scope, name space-scope or static block-scope variable.

Usage

Each copy of an **omp threadprivate** data variable is initialized once prior to first use of that copy. If an object is changed before being used to initialize a **threadprivate** data variable, behavior is unspecified.

A thread must not reference another thread's copy of an **omp threadprivate** data variable. References will always be to the master thread's copy of the data variable when executing serial and master regions of the program.

Use of the **omp threadprivate** directive is governed by the following points:
- An **omp threadprivate** directive must appear at file scope outside of any definition or declaration.
- The **omp threadprivate** directive is applicable to static-block scope variables and may appear in lexical blocks to reference those block-scope variables. The directive must appear in the scope of the variable and not in a nested scope, and must precede all references to variables in its list.
- A data variable must be declared with file scope prior to inclusion in an **omp threadprivate** directive *list*.
- An **omp threadprivate** directive and its *list* must lexically precede any reference to a data variable found in that *list*.
- A data variable specified in an **omp threadprivate** directive in one translation unit must also be specified as such in all other translation units in which it is declared.
- Data variables specified in an **omp threadprivate** *list* must not appear in any clause other than the **copyin**, **copyprivate**, **if**, **num_threads**, and **schedule** clauses.
- The address of a data variable in an **omp threadprivate** *list* is not an address constant.
- A data variable specified in an **omp threadprivate** *list* must not have an incomplete or reference type.

#pragma omp task
Purpose

The OpenMP **task** pragma can be used to explicitly define a task.

Use the **task** pragma when you want to identify a block of code to be executed in parallel with the code outside the task region. The **task** pragma can be useful for

parallelizing irregular algorithms such as pointer chasing or recursive algorithms for which other OpenMP workshare constructs are inadequate. The **task** directive only takes effect if you specify the **-qsmp** compiler option.

Syntax

```
►►──#──pragma──omp task──┬─clause─┬──────────────────────────►◄
                         └───,◄───┘
```

Parameters

clause is any of the following:

if (*exp***)**
> When the `if` argument is specified, the program code executes in parallel only if the scalar expression represented by *exp* evaluates to a nonzero value at run time. Only one `if` clause can be specified.

private (*list***)**
> Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

firstprivate (*list***)**
> Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized with the value of the original variable as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

default

untied
> When a task region is suspended, untied tasks can be resumed by any thread in a team.

shared (*list***)**
> Declares the scope of the comma-separated data variables in *list* to be shared across all threads.

Related reference

"#pragma omp taskwait"

#pragma omp taskwait
Purpose

Use the **taskwait** pragma to specify a *wait* for child tasks to be completed that are generated by the current task.

Syntax

```
►►──#──pragma──omp taskwait───────────────────────────────────►◄
```

Related reference

"#pragma omp task" on page 389

Chapter 6. Compiler predefined macros

Predefined macros can be used to conditionally compile code for specific compilers, specific versions of compilers, specific environments and/or specific language features.

Predefined macros fall into several categories:
- "General macros"
- "Macros related to the platform" on page 393
- "Macros related to compiler features" on page 394

"Examples of predefined macros" on page 402 show how you can use them in your code.

General macros

The following predefined macros are always predefined by the compiler. Unless noted otherwise, all the following macros are *protected*, which means that the compiler will issue a warning if you try to undefine or redefine them.

Table 40. General predefined macros

Predefined macro name	Description	Predefined value
__BASE_FILE__	Indicates the name of the primary source file.	The fully qualified file name of the primary source file.
__DATE__	Indicates the date that the source file was preprocessed.	A character string containing the date when the source file was preprocessed.
__FILE__	Indicates the name of the preprocessed source file.	A character string containing the name of the preprocessed source file.
__FUNCTION__	Indicates the name of the function currently being compiled.	A character string containing the name of the function currently being compiled.
__LINE__	Indicates the current line number in the source file.	An integer constant containing the line number in the source file.
__SIZE_TYPE__	Indicates the underlying type of `size_t` on the current platform. Not protected.	`unsigned int` in 32-bit compilation mode and `unsigned long` in 64-bit compilation mode.
__TIME__	Indicates the time that the source file was preprocessed.	A character string containing the time when the source file was preprocessed.

Table 40. General predefined macros (continued)

Predefined macro name	Description	Predefined value
__TIMESTAMP__	Indicates the date and time when the source file was last modified. The value changes as the compiler processes any include files that are part of your source program.	A character string literal in the form "*Day Mmm dd hh:mm:ss yyyy*", where:: *Day* Represents the day of the week (Mon, Tue, Wed, Thu, Fri, Sat, or Sun). *Mmm* Represents the month in an abbreviated form (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec). *dd* Represents the day. If the day is less than 10, the first d is a blank character. *hh* Represents the hour. *mm* Represents the minutes. *ss* Represents the seconds. *yyyy* Represents the year.

Macros indicating the XL C/C++ compiler product

Macros related to the XL C/C++ compiler are always predefined, and are protected (the compiler will issue a warning if you try to undefine or redefine them).

Table 41. Compiler product predefined macros

Predefined macro name	Description	Predefined value
▶ C ◀ __IBMC__	Indicates the level of the XL C compiler.	An integer in the format *VRM*, where : *V* Represents the version number *R* Represents the release number *M* Represents the modification number In XL C/C++ V11.1, the value of the macro is 1110.
▶ C++ ◀ __IBMCPP__	Indicates the level of the XL C++ compiler.	An integer in the format *VRM*, where : *V* Represents the version number *R* Represents the release number *M* Represents the modification number In XL C/C++ V11.1, the value of the macro is 1110.
▶ C ◀ __xlc__	Indicates the level of the XL C compiler.	A string in the format "*V.R.M.F*", where: *V* Represents the version number *R* Represents the release number *M* Represents the modification number *F* Represents the fix level In XL C/C++ V11.1, the value of the macro is "11.1.0.0".

Table 41. Compiler product predefined macros (continued)

Predefined macro name	Description	Predefined value
__xlC__	Indicates the level of the XL C++ compiler. Using the XL C compiler also automatically defines this macro.	A four-digit hexadecimal integer in the format 0x*VVRR*, where: *V* Represents the version number *R* Represents the release number In XL C/C++ V11.1, the value of the macro is 0x0b01.

Macros related to the platform

The following predefined macros are provided to facilitate porting applications between platforms. All platform-related predefined macros are unprotected and may be undefined or redefined without warning unless otherwise specified.

Table 42. Platform-related predefined macros

Predefined macro name	Description	Predefined value	Predefined under the following conditions
_BIG_ENDIAN, __BIG_ENDIAN__	Indicates that the platform is big-endian (that is, the most significant byte is stored at the memory location with the lowest address).	1	Always predefined.
__ELF__	Indicates that the ELF object model is in effect.	1	Always predefined for the Linux platform.
C++ __GXX_WEAK__	Indicates that weak symbols are supported (used for template instantiation by the linker).	1	Always predefined.
__HOS_LINUX__	Indicates that the host operating system is Linux. Protected.	1	Always predefined for all Linux platforms.
_ILP32, __ILP32__	Indicates that the target platform uses 32-bit `int`, `long int`, and pointer types.	1	Predefined when the target platform uses 32-bit `int`, `long int`, and pointer types.
__linux, __linux__	Indicates that the platform is Linux.	1	Always predefined for all Linux platforms.
_LP64, __LP64__	Indicates that the target platform uses 64-bit `long int` and pointer types, and a 32-bit `int` type.	1	Predefined when the target platform uses 64-bit `long int` and pointer types, and 32-bit a `int` type.
__powerpc, __powerpc__	Indicates that the target is a Power architecture.	1	Predefined when the target is a Power architecture.
__powerpc64__	Indicates that the target is a Power architecture and that 64-bit compilation mode is enabled.	1	Predefined when the target is a Power architecture and **-q64** is in effect.
__PPC, __PPC__	Indicates that the target is a Power architecture.	1	Predefined when the target is a Power architecture.
__PPC64__	Indicates that the target is a Power architecture and that 64-bit compilation mode is enabled.	1	Predefined when the target is a Power architecture and **-q64** is in effect.
__THW_PPC__	Indicates that the target is a Power architecture.	1	Predefined when the target is a Power architecture.

Table 42. Platform-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined under the following conditions
__TOS_LINUX__	Indicates that the target operating system is Linux.	1	Predefined when the target is a Power architecture.
__unix, __unix__	Indicates that the operating system is a variety of UNIX.	1	Always predefined.

Macros related to compiler features

Feature-related macros are predefined according to the setting of specific compiler options or pragmas. Unless noted otherwise, all feature-related macros are protected (the compiler will issue a warning if you try to undefine or redefine them).

Feature-related macros are discussed in the following sections:
- "Macros related to compiler option settings"
- "Macros related to architecture settings" on page 396
- "Macros related to language levels" on page 397

Macros related to compiler option settings

The following macros can be tested for various features, including source input characteristics, output file characteristics, optimization, and so on. All of these macros are predefined by a specific compiler option or suboption, or any invocation or pragma that implies that suboption. If the suboption enabling the feature is not in effect, then the macro is undefined.

Table 43. General option-related predefined macros

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
__ALTIVEC__	Indicates support for vector data types. (unprotected)	1	-qaltivec
__64BIT__	Indicates that 64-bit compilation mode is in effect.	1	-q64
_CHAR_SIGNED, __CHAR_SIGNED__	Indicates that the default character type is `signed char`.	1	-qchars=signed
_CHAR_UNSIGNED, __CHAR_UNSIGNED__	Indicates that the default character type is `unsigned char`.	1	-qchars=unsigned
▶ C++ __EXCEPTIONS	Indicates that C++ exception handling is enabled.	1	-qeh

Table 43. General option-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
__IBM_GCC_ASM	Indicates support for GCC inline `asm` statements.	1	► **C** -qasm=gcc and -qlanglvl=extc99 | extc89 | extended or-qkeyword=asm ► **C++** -qasm=gcc and-qlanglvl=extended
		0	► **C** -qnoasm and -qlanglvl=extc99 | extc89 | extended or-qkeyword=asm ► **C++** -qnoasm and-qlanglvl=extended
► **C++** __IBM_STDCPP_ASM	Indicates that support for GCC inline `asm` statements is disabled.	0	**-qnoasm=stdcpp**
► **C++** __IGNERRNO__	Indicates that system calls do not modify errno, thereby enabling certain compiler optimizations.	1	-qignerrno
► **C++** __INITAUTO__	Indicates the value to which automatic variables which are not explicitly initialized in the source program are to be initialized.	The two-digit hexadecimal value specified in the **-qinitauto** compiler option.	-qinitauto=*hex value*
► **C++** __INITAUTO_W__	Indicates the value to which automatic variables which are not explicitly initialized in the source program are to be initialized.	An eight-digit hexadecimal corresponding to the value specified in the **-qinitauto** compiler option repeated 4 times.	-qinitauto=*hex value*
► **C++** __LIBANSI__	Indicates that calls to functions whose names match those in the C Standard Library are in fact the C library functions, enabling certain compiler optimizations.	1	-qlibansi
__LONGDOUBLE64	Indicates that the size of a `long double` type is 64 bits.	1	**-qnoldbl128**

Table 43. General option-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
__LONGDOUBLE128, __LONG_DOUBLE_128__	Indicates that the size of a `long double` type is 128 bits.	1	-qldbl128
__OPTIMIZE__	Indicates the level of optimization in effect.	2	-O │ -O2
		3	-O3 │ -O4 │ -O5
__OPTIMIZE_SIZE__	Indicates that optimization for code size is in effect.	1	-O │ -O2 │ -O3 │ -O4 │ -O5 and -qcompact
▶ C++ __RTTI_DYNAMIC_CAST__	Indicates that runtime type identification information for the `dynamic_cast` operator is generated.	1	-qrtti
▶ C++ __RTTI_TYPE_INFO__	Indicates that runtime type identification information for the `typeid` operator is generated.	1	-qrtti
▶ C++ __NO_RTTI__	Indicates that runtime type identification information is disabled.	1	-qnortti
▶ C++ __TEMPINC__	Indicates that the compiler is using the template-implementation file method of resolving template functions.	1	-qtempinc
__VEC__	Indicates support for vector data types.	10205	-qaltivec

Macros related to architecture settings

The following macros can be tested for target architecture settings. All of these macros are predefined to a value of 1 by a **-qarch** compiler option setting, or any other compiler option that implies that setting. If the **-qarch** suboption enabling the feature is not in effect, then the macro is undefined.

*Table 44. **-qarch**-related macros*

Macro name	Description	Predefined by the following -qarch suboptions
_ARCH_COM	Indicates that the application is targeted to run on any Power processor.	Defined for all **-qarch** suboptions except **auto**.
_ARCH_PPC	Indicates that the application is targeted to run on any Power processor.	Defined for all **-qarch** suboptions except **auto**.
_ARCH_PPC64	Indicates that the application is targeted to run on Power processors with 64-bit support.	ppc64 │ pwr3 │ rs64b │ rs64c │ ppc64gr │ ppc64grsq │ ppc64v │ pwr4 │ pwr5 │ pwr5x │ pwr6 │ pwr6e │ pwr7 │ ppc970
_ARCH_PPCGR	Indicates that the application is targeted to run on Power processors with graphics support.	ppcgr │ pwr3 │ rs64b │ rs64c │ ppc64gr │ ppc64grsq │ ppc64v │ pwr4 │ pwr5 │ pwr5x │ pwr6 │ pwr6e │ pwr7 │ ppc970

Table 44. -qarch-related macros (continued)

Macro name	Description	Predefined by the following -qarch suboptions
_ARCH_PPC64GR	Indicates that the application is targeted to run on Power processors with 64-bit and graphics support.	pwr3 | rs64b | rs64c | ppc64gr | ppc64v | pwr4 | pwr5 | pwr5x | pwr6 | pwr6e | pwr7 | ppc970
_ARCH_PPC64GRSQ	Indicates that the application is targeted to run on Power processors with 64-bit, graphics, and square root support.	pwr3 | rs64b | rs64c | ppc64grsq | ppc64v | pwr4 | pwr5 | pwr5x | pwr6 | pwr6e | pwr7 | ppc970
_ARCH_PPC64V	Indicates that the application is targeted to run on Power processors with 64-bit and vector processing support.	ppc64v | ppc970 | pwr6 | pwr6e | pwr7
_ARCH_PPC970	Indicates that the application is targeted to run on the PowerPC 970 processor.	ppc970
_ARCH_PWR3	Indicates that the application is targeted to run on POWER3 processors.	pwr3 | pwr4 | pwr5 | pwr5x | pwr6 | pwr6e | pwr7 | ppc970
_ARCH_PWR4	Indicates that the application is targeted to run on POWER4 processors.	pwr4 | pwr5 | pwr5x | pwr6 | pwr6e | pwr7 | ppc970
_ARCH_PWR5	Indicates that the application is targeted to run on POWER5 processors.	pwr5 | pwr5x | pwr6 | pwr6e | pwr7
_ARCH_PWR5X	Indicates that the application is targeted to run on POWER5+ processors.	pwr5x | pwr6 | pwr6e | pwr7
_ARCH_PWR6	Indicates that the application is targeted to run on POWER6 processors.	pwr6 | pwr6e | pwr7
_ARCH_PWR6E	Indicates that the application is targeted to run on POWER6 processors running in POWER6 raw mode.	pwr6e
_ARCH_PWR7	Indicates that the application is targeted to run on POWER7 processors.	pwr7
_ARCH_RS64B	Indicates that the application is targeted to run on the RS64II processor.	rs64b
_ARCH_RS64C	Indicates that the application is targeted to run on the RS64III processor.	rs64c

Macros related to language levels

The following macros can be tested for C99 features, features related to GNU C or C++, and other IBM language extensions. All of these macros are predefined to a value of 1 by a specific language level, represented by a suboption of the **-qlanglvl** compiler option, or any invocation or pragma that implies that suboption. If the suboption enabling the feature is not in effect, then the macro is undefined. For descriptions of the features related to these macros, see the *XL C/C++ Language Reference*.

Table 45. Predefined macros for language features

Predefined macro name	Description	Predefined when the following language level is in effect
▶ C++ __BOOL__	Indicates that the bool keyword is accepted.	Always defined except when **-qnokeyword=bool** is in effect.
▶ C __C99_BOOL	Indicates support for the _Bool data type.	stdc99 | extc99 | extc89 | extended

Table 45. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
▶ C �__C99_COMPLEX	Indicates support for complex data types.	stdc99 | extc99 | extc89 | extended
▶ C �__C99_CPLUSCMT	Indicates support for C++ style comments	stdc99 | extc99 | stdc89 | extc89 | extended (also **-qcpluscmt**)
__C99_COMPOUND_LITERAL	Indicates support for compound literals.	▶ C stdc99 | extc99 | extc89 | extended ▶ C++ extended | extended0x
▶ C �__C99_DESIGNATED_INITIALIZER	Indicates support for designated initialization.	stdc99 | extc99 | extc89 | extended
▶ C �__C99_DUP_TYPE_QUALIFIER	Indicates support for duplicated type qualifiers.	stdc99 | extc99 | extc89 | extended
▶ C �__C99_EMPTY_MACRO_ARGUMENTS	Indicates support for empty macro arguments.	stdc99 | extc99 | extc89 | extended
▶ C �__C99_FLEXIBLE_ARRAY_MEMBER	Indicates support for flexible array members.	stdc99 | extc99 | extc89 | extended
__C99__FUNC__	Indicates support for the __func__ predefined identifier.	▶ C stdc99 | extc99 | extc89 | extended ▶ C++ extended | extended0x |c99__func__
__C99_HEX_FLOAT_CONST	Indicates support for hexadecimal floating constants.	▶ C stdc99 | extc99 | extc89 | extended ▶ C++ extended | extended0x | c99hexfloat
▶ C �__C99_INLINE	Indicates support for the inline function specifier.	stdc99 | extc99 (also **-qkeyword=inline**)
__C99_LLONG	Indicates support for C99-style long long data types and literals.	▶ C stdc99 | extc99 ▶ C++ extended0x | c99longlong
__C99_MACRO_WITH_VA_ARGS	Indicates support for function-like macros with variable arguments.	▶ C stdc99 | extc99 | extc89 | extended ▶ C++ extended | extended0x | varargmacros
__C99_MAX_LINE_NUMBER	Indicates that the maximum line number is 2147483647.	▶ C stdc99 | extc99 | extc89 | extended ▶ C++ extended0x | c99preprocessor
▶ C �__C99_MIXED_DECL_AND_CODE	Indicates support for mixed declaration and code.	stdc99 | extc99 | extc89 | extended

Table 45. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
__C99_MIXED_STRING_CONCAT	Indicates support for concatenation of wide string and non-wide string literals.	`C` stdc99 | extc99 | extc89 | extended `C++` extended0x | c99preprocessor
`C` __C99_NON_LVALUE_ARRAY_SUB	Indicates support for non-lvalue subscripts for arrays.	stdc99 | extc99 | extc89 | extended
`C` __C99_NON_CONST_AGGR_INITIALIZER	Indicates support for non-constant aggregate initializers.	stdc99 | extc99 | extc89 | extended
__C99_PRAGMA_OPERATOR	Indicates support for the `_Pragma` operator.	`C` stdc99 | extc99 | extc89 | extended `C++` extended extended0x
`C` __C99_REQUIRE_FUNC_DECL	Indicates that implicit function declaration is not supported.	stdc99
__C99_RESTRICT	Indicates support for the C99 `restrict` qualifier.	`C` stdc99 | extc99 (also **-qkeyword=restrict**) `C++` extended | extended0x (also **-qkeyword=restrict**)
`C` __C99_STATIC_ARRAY_SIZE	Indicates support for the `static` keyword in array parameters to functions.	stdc99 | extc99 | extc89 | extended
`C` __C99_STD_PRAGMAS	Indicates support for standard pragmas.	stdc99 | extc99 | extc89 | extended
`C` __C99_TGMATH	Indicates support for type-generic macros in tgmath.h	stdc99 | extc99 | extc89 | extended
__C99_UCN	Indicates support for universal character names.	`C` stdc99 | extc99 | ucs `C++` ucs
`C` __C99_VAR_LEN_ARRAY	Indicates support for variable length arrays.	stdc99 | extc99 | extc89 | extended
`C++` __C99_VARIABLE_LENGTH_ARRAY	Indicates support for variable length arrays.	extended | extended0x | c99vla
__DIGRAPHS__	Indicates support for digraphs.	`C` stdc99 | extc99 | extc89 | extended (also **-qdigraph**) `C++` extended | extended0x | compat366 | strict98(also **-qdigraph**)

Table 45. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
`C` __EXTENDED__	Indicates that language extensions are supported.	extended
`C++` __IBM__ALIGN	Indicates support for the `__align` specifier.	Always defined except when **-qnokeyword=__alignof** is specified
__IBM__ALIGNOF__	Indicates support for the `__alignof__` operator.	`C` extc99 | extc89 | extended `C++` extended
__IBM_ATTRIBUTES	Indicates support for type, variable, and function attributes.	`C` extc99 | extc89 | extended `C++` extended | extended0x
__IBM_COMPUTED_GOTO	Indicates support for computed `goto` statements.	`C` extc99 | extc89 | extended `C++` extended | extended0x | gnu_computedgoto
__IBM_EXTENSION_KEYWORD	Indicates support for the `__extension__` keyword.	`C` extc99 | extc89 | extended `C++` extended | extended0x | compat366 | strict98
`C` __IBM_GCC__INLINE__	Indicates support for the GCC `__inline__` specifier.	extc99 | extc89 | extended
`C` __IBM_DOLLAR_IN_ID	Indicates support for dollar signs in identifiers.	extc99 | extc89 | extended
`C` __IBM_GENERALIZED_LVALUE	Indicates support for generalized lvalues.	extc99 | extc89 | extended
__IBM_INCLUDE_NEXT	Indicates support for the `#include_next` preprocessing directive.	`C` Always defined `C++` Always defined except when **-qlanglvl=nognu_include_next** is in effect.
__IBM_LABEL_VALUE	Indicates support for labels as values.	`C` extc99 | extc89 | extended `C++` extended | extended0x |gnu_labelvalue

Table 45. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
__IBM_LOCAL_LABEL	Indicates support for local labels.	▶ **C** extc99 | extc89 | extended ▶ **C++** extended | extended0x | gnu_locallabel
__IBM_MACRO_WITH_VA_ARGS	Indicates support for variadic macro extensions.	▶ **C** extc99 | extc89 | extended ▶ **C++** extended | extended0x | gnu_varargmacros
▶ **C** _IBM_NESTED_FUNCTION	Indicates support for nested functions.	extc99 | extc89 | extended
▶ **C** __IBM_PP_PREDICATE	Indicates support for `#assert`, `#unassert`, `#cpu`, `#machine`, and `#system` preprocessing directives.	extc99 | extc89 | extended
▶ **C** __IBM_PP_WARNING	Indicates support for the `#warning` preprocessing directive.	extc99 | extc89 | extended
▶ **C** __IBM_REGISTER_VARS	Indicates support for variables in specified registers.	Always defined.
__IBM__TYPEOF__	Indicates support for the `__typeof__` or `typeof` keyword.	▶ **C** always defined ▶ **C++** extended | extended0x (Also **-qkeyword=typeof**)
▶ **C++0x** __IBMCPP_AUTO_TYPEDEDUCTION	Indicates support for the auto type deduction feature.	▶ **C++** extended0x | autotypededuction
▶ **C++0x** __IBMCPP_C99_LONG_LONG	Indicates support for the C99 `long long` feature.	▶ **C++** extended0x | c99longlong
▶ **C++0x** __IBMCPP_C99_PREPROCESSOR	Indicates support for the C99 preprocessor features adopted in the C++0x standard.	▶ **C++** extended0x | c99preprocessor
▶ **C++0x** __IBMCPP_DECLTYPE	Indicates support for the decltype feature.	▶ **C++** extended0x | decltype
▶ **C++0x** __IBMCPP_DELEGATING_CTORS	Indicates support for the delegating constructors feature.	▶ **C++** extended0x | delegatingctors
▶ **C++0x** __IBMCPP_EXTENDED_FRIEND	Indicates support for the extended friend declarations feature.	▶ **C++** extended0x | extendedfriend
▶ **C++0x** __IBMCPP_EXTERN_TEMPLATE	Indicates support for the explicit instantiation declarations feature.	▶ **C++** extended | extended0x | externtemplate
▶ **C++0x** __IBMCPP_INLINE_NAMESPACE	Indicates support for the inline namespace definitions feature.	▶ **C++** extended0x | inlinenamespace

Table 45. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
► C++0x __IBMCPP_STATIC_ASSERT	Indicates support for the static assertions feature.	► C++ extended0x | static_assert
► C++0x __IBMCPP_VARIADIC_TEMPLATES	Indicates support for the variadic templates feature.	► C++ extended0x | variadic[templates]
_LONG_LONG	Indicates support for long long data types.	► C stdc99 | extc99 | | stdc89 | extc89 | extended (also -qlonglong) ► C++ extended0x | c99longlong | extended (also **-qlonglong**)
► C __STDC__	Indicates that the compiler conforms to the ANSI/ISO C standard.	Predefined to 1 if ANSI/ISO C standard conformance is in effect.
__STDC_HOSTED__	Indicates that the implementation is a hosted implementation of the ANSI/ISO C standard. (That is, the hosted environment has all the facilities of the standard C available).	► C stdc99 | extc99 ► C++ extended0x
► C __STDC_VERSION__	Indicates the version of ANSI/ISO C standard which the compiler conforms to.	The format is yyyymmL. (For example, the format is 199901L for C99.)

Examples of predefined macros

This example illustrates use of the __FUNCTION__ and the __C99__FUNC__ macros to test for the availability of the C99 __func__ identifier to return the current function name:

```
#include <stdio.h>

#if defined(__C99__FUNC__)
#define PRINT_FUNC_NAME() printf (" In function %s \n", __func__);
#elif defined(__FUNCTION__)
#define PRINT_FUNC_NAME() printf (" In function %s \n", __FUNCTION__);
#else
#define PRINT_FUNC_NAME() printf (" Function name unavailable\n");
#endif

void foo(void);

int main(int argc, char **argv)
{
   int k = 1;
   PRINT_FUNC_NAME();
   foo();
   return 0;
}

void foo (void)
```

```
{
   PRINT_FUNC_NAME();
   return;
}
```

The output of this example is:

```
In function main
In function foo
```

▶ C++ This example illustrates use of the __FUNCTION__ macro in a C++
program with virtual functions.

```
#include <stdio.h>
class X { public: virtual void func() = 0;};

class Y : public X {
   public: void func() { printf("In function %s \n", __FUNCTION__);}
};

int main() {
  Y aaa;
  aaa.func();
}
```

The output of this example is:

```
In function Y::func()
```

Chapter 7. Compiler built-in functions

A built-in function is a coding extension to C and C++ that allows a programmer to use the syntax of C function calls and C variables to access the instruction set of the processor of the compiling machine. IBM Power architectures have special instructions that enable the development of highly optimized applications. Access to some Power instructions cannot be generated using the standard constructs of the C and C++ languages. Other instructions can be generated through standard constructs, but using built-in functions allows exact control of the generated code. Inline assembly language programming, which uses these instructions directly, is not fully supported by XL C/C++ and other compilers. Furthermore, the technique can be time-consuming to implement.

As an alternative to managing hardware registers through assembly language, XL C/C++ built-in functions provide access to the optimized Power instruction set and allow the compiler to optimize the instruction scheduling.

▶ `C++` To call any of the XL C/C++ built-in functions in C++, you must include the header file builtins.h in your source code.

The following sections describe the available built-in functions for the Linux platform.
- "Fixed-point built-in functions"
- "Binary floating-point built-in functions" on page 414
- "Synchronization and atomic built-in functions" on page 424
- "Cache-related built-in functions" on page 431
- "Block-related built-in functions" on page 440
- "Miscellaneous built-in functions" on page 500
- "Built-in functions for parallel processing" on page 506

Fixed-point built-in functions

Fixed-point built-in functions are grouped into the following categories:
- "Absolute value functions" on page 406

- "Assert functions" on page 406

- "Count zero functions" on page 407

- "Load functions" on page 409

- "Multiply functions" on page 409

- "Population count functions" on page 410

- "Rotate functions" on page 411

- "Store functions" on page 412

- "Trap functions" on page 413

Absolute value functions

__labs, __llabs
Purpose

Absolute Value Long, Absolute Value Long Long

Returns the absolute value of the argument.

Prototype

signed long __labs (signed long);

signed long long __llabs (signed long long);

Assert functions

__assert1, __assert2
Purpose

Generates trap instructions.

Prototype

int __assert1 (int, int, int);

void __assert2 (int);

Bit permutation functions

__bpermd
Purpose

Byte Permute Doubleword

Returns the result of a bit permutation operation.

Prototype

long long __bpermd (long long bit_selector, long long source);

Usage

Eight bits are returned, each corresponding to a bit within source, and were selected by a byte of bit_selector. If byte i of bit_selector is less than 64, the permuted bit i is set to the bit of source specified by byte i of bit_selector; otherwise the permuted bit i is set to 0. The permuted bits are placed in the least-significant byte of the result value and the remaining bits are filled with 0s.

Valid only when **-qarch** is set to target POWER7 processors in 64-bit mode.

Comparison functions

__cmpb
Purpose

Compare Bytes

Compares each of the eight bytes of *source1* with the corresponding byte of *source2*. If byte *i* of *source1* and byte *i* of *source2* are equal, 0xFF is placed in the corresponding byte of the result; otherwise, 0x00 is placed in the corresponding byte of the result.

Prototype

long long __cmpb (long long source1, long long source2);

Usage

Valid only when **-qarch** is set to target POWER6 and POWER7 processors.

Count zero functions

__cntlz4, __cntlz8
Purpose

Count Leading Zeros, 4/8-byte integer

Prototype

unsigned int __cntlz4 (unsigned int);

unsigned int __cntlz8 (unsigned long long);

__cnttz4, __cnttz8
Purpose

Count Trailing Zeros, 4/8-byte integer

Prototype

unsigned int __cnttz4 (unsigned int);

unsigned int __cnttz8 (unsigned long long);

Division functions

These division functions are valid only when **-qarch** is set to target POWER7 processors.

__divde
Purpose

Divide Doubleword Extended

Returns the result of a doubleword extended division. The result has a value equal to *dividend/divisor*.

Prototype

> long long __divde (long long dividend, long long divisor);

Usage

Valid only when **-qarch** is set to target POWER7 processors in 64-bit mode.

Note: If the result of the division is larger than 32 bits or if the divisor is 0, the return value of the function is undefined.

__divdeu
Purpose

Divide Doubleword Extended Unsigned

Returns the result of a double word extended unsigned division. The result has a value equal to *dividend/divisor*.

Prototype

> unsigned long long __divdeu (unsigned long long dividend, unsigned long long divisor);

Usage

Valid only when **-qarch** is set to target POWER7 processors in 64-bit mode.

Note: If the result of the division is larger than 32 bits or if the divisor is 0, the return value of the function is undefined.

__divwe
Purpose

Divide Word Extended

Returns the result of a word extended division. The result has a value equal to *dividend/divisor*.

Prototype

> int __divwe(int dividend, int divisor);

Usage

Valid only when **-qarch** is set to target POWER7 processors.

Note: If the divisor is 0, the return value of the function is undefined.

__divweu
Purpose

Divide Word Extended Unsigned

Returns the result of a word extended unsigned division. The result has a value equal to *dividend/divisor*.

Prototype

> unsigned int __divweu(unsigned int dividend, unsigned int divisor);

Usage

Valid only when **-qarch** is set to target POWER7 processors.

Note: If the divisor is 0, the return value of the function is undefined.

Load functions

__load2r, __load4r
Purpose

Load Halfword Byte Reversed, Load Word Byte Reversed

Prototype

> unsigned short __load2r (unsigned short*);

> unsigned int __load4r (unsigned int*);

__load8r
Purpose

Load with Byte Reversal (8-byte integer)

Performs an eight-byte byte-reversed load from the given address.

Prototype

> unsigned long long __load8r (unsigned long long * address);

Usage

Valid only when **-qarch** and **-q64** are set to target POWER7 processors.

Multiply functions

__mulhd, __mulhdu
Purpose

Multiply High Doubleword Signed, Multiply High Doubleword Unsigned

Returns the highorder 64 bits of the 128bit product of the two parameters.

Prototype

> long long int __mulhd (long int, long int);

> unsigned long long int __mulhdu (unsigned long int, unsigned long int);

Usage

Valid only in 64-bit mode.

__mulhw, __mulhwu
Purpose

Multiply High Word Signed, Multiply High Word Unsigned

Returns the highorder 32 bits of the 64bit product of the two parameters.

Prototype

int __mulhw (int, int);

unsigned int __mulhwu (unsigned int, unsigned int);

Population count functions

__popcnt4, __popcnt8
Purpose

Population Count, 4/8-byte integer

Returns the number of bits set for a 32/64-bit integer.

Prototype

int __popcnt4 (unsigned int);

int __popcnt8 (unsigned long long);

__popcntb
Purpose

Population Count Byte

Counts the 1 bits in each byte of the parameter and places that count into the corresponding byte of the result.

Prototype

unsigned long __popcntb(unsigned long);

__poppar4, __poppar8
Purpose

Population Parity, 4/8-byte integer

Checks whether the number of bits set in a 32/64-bit integer is an even or odd number.

Prototype

int __poppar4(unsigned int);

int __poppar8(unsigned long long);

Return value

Returns 1 if the number of bits set in the input parameter is odd. Returns 0 otherwise.

Rotate functions

__rdlam
Purpose

Rotate Double Left and AND with Mask

Rotates the contents of *rs* left *shift* bits, and ANDs the rotated data with the *mask*.

Prototype

> unsigned long long __rdlam (unsigned long long *rs*, unsigned int *shift*, unsigned long long *mask*);

Parameters

mask
    Must be a constant that represents a contiguous bit field.

__rldimi, __rlwimi
Purpose

Rotate Left Doubleword Immediate then Mask Insert, Rotate Left Word Immediate then Mask Insert

Rotates *rs* left *shift* bits then inserts *rs* into *is* under bit mask *mask*.

Prototype

> unsigned long long __rldimi (unsigned long long *rs*, unsigned long long *is*, unsigned int *shift*, unsigned long long *mask*);

> unsigned int __rlwimi (unsigned int *rs*, unsigned int *is*, unsigned int *shift*, unsigned int *mask*);

Parameters

shift
    A constant value 0 to 63 (`__rldimi`) or 31 (`__rlwimi`).
mask
    Must be a constant that represents a contiguous bit field.

__rlwnm
Purpose

Rotate Left Word then AND with Mask

Rotates *rs* left *shift* bits, then ANDs *rs* with bit mask *mask*.

Prototype

> unsigned int __rlwnm (unsigned int *rs*, unsigned int *shift*, unsigned int *mask*);

Parameters

mask
  Must be a constant that represents a contiguous bit field.

__rotatel4, __rotatel8
Purpose

Rotate Left Word, Rotate Left Doubleword

Rotates *rs* left *shift* bits.

Prototype

> unsigned int __rotatel4 (unsigned int *rs*, unsigned int *shift*);

> unsigned long long __rotatel8 (unsigned long long *rs*, unsigned long long *shift*);

Store functions

__store2r, __store4r
Purpose

Store 2/4-byte Reversal

Prototype

> void __store2r (unsigned short, unsigned short*);

> void __store4r (unsigned int, unsigned int*);

__store8r
Purpose

Store with Byte-Reversal (eight-byte integer)

Takes the loaded eight-byte integer value and performs a byte-reversed store operation.

Prototype

> void __store8r (unsigned long long source, unsigned long long * address);

Usage

Valid only when **-qarch** is set to target POWER7 processors in 64-bit mode.

Trap functions

__tdw, __tw
Purpose

Trap Doubleword, Trap Word

Compares parameter *a* with parameter *b*. This comparison results in five conditions which are ANDed with a 5-bit constant *TO*. If the result is not 0 the system trap handler is invoked.

Prototype

void __tdw (long *a*, long *b*, unsigned int *TO*);

void __tw (int *a*, int *b*, unsigned int *TO*);

Parameters

TO
:   A value of 0 to 31 inclusive. Each bit position, if set, indicates one or more of the following possible conditions:

    0 (high-order bit)
    :   *a* is less than *b*, using signed comparison.

    1
    :   *a* is greater than *b*, using signed comparison.

    2
    :   *a* is equal to *b*

    3
    :   *a* is less than *b*, using unsigned comparison.

    4 (low-order bit)
    :   *a* is greater than *b*, using unsigned comparison.

Usage

__tdw is valid only in 64-bit mode.

__trap, __trapd
Purpose

Trap if the Parameter is not Zero, Trap if the Parameter is not Zero Doubleword

Prototype

void __trap (int);

void __trapd (long);

Usage

__trapd is valid only in 64-bit mode.

Binary floating-point built-in functions

Floating-point built-in functions are grouped into the following categories:

- "Absolute value functions" on page 406
- "Conversion functions"
- "FPSCR functions" on page 417
- "Multiply-add/subtract functions" on page 419
- "Reciprocal estimate functions" on page 420
- "Rounding functions" on page 420
- "Select functions" on page 422
- "Square root functions" on page 422
- "Software division functions" on page 422

Absolute value functions

__fabss
Purpose

Floating Absolute Value Single

Returns the absolute value of the argument.

Prototype

float __fabss (float);

__fnabs
Purpose

Floating Negative Absolute Value, Floating Negative Absolute Value Single

Returns the negative absolute value of the argument.

Prototype

double __fnabs (double);

float __fnabss (float);

Conversion functions

__cmplx, __cmplxf, __cmplxl
Purpose

Converts two real parameters into a single complex value.

Prototype

double _Complex __cmplx (double, double);

float _Complex __cmplxf (float, float);

long double _Complex __cmplxl (long double, long double);

__fcfid

Purpose

Floating Convert from Integer Doubleword

Converts a 64-bit signed integer stored in a double to a double-precision floating-point value.

Prototype

    double __fcfid (double);

__fcfud

Purpose

Floating-point Conversion from Unsigned integer Double word

Converts a 64-bit unsigned integer stored in a double into a double-precision floating-point value.

Prototype

    double __fcfud(double);

__fctid

Purpose

Floating Convert to Integer Doubleword

Converts a double-precision argument to a 64-bit signed integer, using the current rounding mode, and returns the result in a double.

Prototype

    double __fctid (double);

__fctidz

Purpose

Floating Convert to Integer Doubleword with Rounding towards Zero

Converts a double-precision argument to a 64-bit signed integer, using the rounding mode round-toward-zero, and returns the result in a double.

Prototype

    double __fctidz (double);

__fctiw

Purpose

Floating Convert to Integer Word

Converts a double-precision argument to a 32-bit signed integer, using the current rounding mode, and returns the result in a double.

Prototype

> double __fctiw (double);

__fctiwz
Purpose

Floating Convert to Integer Word with Rounding towards Zero

Converts a double-precision argument to a 32-bit signed integer, using the rounding mode round-toward-zero, and returns the result in a double.

Prototype

> double __fctiwz (double);

__fctudz
Purpose

Floating-point Conversion to Unsigned integer Double word with rounding towards Zero

Converts a floating-point value to unsigned integer double word and rounds to zero.

Prototype

> double __fctudz(double);

Result value

The result is a double number, which is rounded to zero.

__fctuwz
Purpose

Floating-point Conversion to Unsigned integer Word with rounding to Zero

Converts a floating-point number into a 32-bit unsigned integer and rounds to zero.

Prototype

> double __fctuwz(double);

Result value

The result is a double number, which is rounded to zero.

__ibm2gccldbl, __ibm2gccldbl_cmplx
Purpose

Converts IBM-style long double data types to GCC long doubles.

Prototype

> long double __ibm2gccldbl (long double);

> _Complex long double __ibm2gccldbl_cmplx (_Complex long double);

Return value

The translated result conforms to GCC requirements for long doubles. However, long double computations performed in IBM-compiled code may not produce bitwise identical results to those obtained purely by GCC.

FPSCR functions

__mtfsb0
Purpose

Move to Floating Point Status/Control Register (FPSCR) Bit 0

Sets bit *bt* of the FPSCR to 0.

Prototype

> void __mtfsb0 (unsigned int *bt*);

Parameters

bt   Must be a constant with a value of 0 to 31.

__mtfsb1
Purpose

Move to FPSCR Bit 1

Sets bit *bt* of the FPSCR to 1.

Prototype

> void __mtfsb1 (unsigned int *bt*);

Parameters

bt   Must be a constant with a value of 0 to 31.

__mtfsf
Purpose

Move to FPSCR Fields

Places the contents of *frb* into the FPSCR under control of the field mask specified by *flm*. The field mask *flm* identifies the 4bit fields of the FPSCR affected.

Prototype

> void __mtfsf (unsigned int *flm*, unsigned int *frb*);

Parameters

flm
  Must be a constant 8-bit mask.

__mtfsfi
Purpose

Move to FPSCR Field Immediate

Places the value of *u* into the FPSCR field specified by *bf*.

Prototype

    void __mtfsfi (unsigned int *bf*, unsigned int *u*);

Parameters

bf Must be a constant with a value of 0 to 7.

u   Must be a constant with a value of 0 to 15.

__readflm
Purpose

Returns a 64-bit double precision floating point, whose 32 low order bits contain the contents of the FPSCR. The 32 low order bits are bits 32 - 63 counting from the highest order bit.

Prototype

    double __readflm (void);

__setflm
Purpose

Takes a double precision floating point number and places the lower 32 bits in the FPSCR. The 32 low order bits are bits 32 - 63 counting from the highest order bit. Returns the previous contents of the FPSCR.

Prototype

    double __setflm (double);

__setrnd
Purpose

Sets the rounding mode.

Prototype

    double __setrnd (int *mode*);

Parameters

The allowable values for *mode* are:
- 0 — round to nearest

- 1 — round to zero
- 2 — round to +infinity
- 3 — round to -infinity

Multiply-add/subtract functions

__fmadd, __fmadds
Purpose

Floating Multiply-Add, Floating Multiply-Add Single

Multiplies the first two arguments, adds the third argument, and returns the result.

Prototype

double __fmadd (double, double, double);

float __fmadds (float, float, float);

__fmsub, __fmsubs
Purpose

Floating Multiply-Subtract, Floating Multiply-Subtract Single

Multiplies the first two arguments, subtracts the third argument and returns the result.

Prototype

double __fmsub (double, double, double);

float __fmsubs (float, float, float);

__fnmadd, __fnmadds
Purpose

Floating Negative Multiply-Add, Floating Negative Multiply-Add Single

Multiplies the first two arguments, adds the third argument, and negates the result.

Prototype

double __fnmadd (double, double, double);

float __fnmadds (float, float, float);

__fnmsub, __fnmsubs
Purpose

Floating Negative Multiply-Subtract

Multiplies the first two arguments, subtracts the third argument, and negates the result.

Prototype

> double __fnmsub (double, double, double);

> float __fnmsubs (float, float, float);

Reciprocal estimate functions

See also "Square root functions" on page 422.

__fre, __fres
Purpose

Floating Reciprocal Estimate, Floating Reciprocal Estimate Single

Prototype

> float __fre (double);

> float __fres (float);

Usage

__fre is valid only when **-qarch** is set to target POWER5 or later processors.

Rounding functions

__fric
Purpose

Floating-point Rounding to Integer with Current rounding mode

Rounds a double-precision floating-point value to integer with the current rounding mode.

Prototype

> double __fric(double);

__frim, __frims
Purpose

Floating Round to Integer Minus

Rounds the floating-point argument to an integer using round-to-minus-infinity mode, and returns the value as a floating-point value.

Prototype

> double __frim (double);

> float __frims (float);

Usage

Valid only when **-qarch** is set to target POWER5+ or later processors.

__frin, __frins
Purpose

Floating Round to Integer Nearest

Rounds the floating-point argument to an integer using round-to-nearest mode, and returns the value as a floating-point value.

Prototype

double __frin (double);

float __frins (float);

Usage

Valid only when **-qarch** is set to target POWER5+ or later processors.

__frip, __frips
Purpose

Floating Round to Integer Plus

Rounds the floating-point argument to an integer using round-to-plus-infinity mode, and returns the value as a floating-point value.

Prototype

double __frip (double);

float __frips (float);

Usage

Valid only when **-qarch** is set to target POWER5+ or later processors.

__friz, __frizs
Purpose

Floating Round to Integer Zero

Rounds the floating-point argument to an integer using round-to-zero mode, and returns the value as a floating-point value.

Prototype

double __friz (double);

float __frizs (float);

Usage

Valid only when **-qarch** is set to target POWER5+ or later processors.

Select functions

__fsel, __fsels
Purpose

Floating Select, Floating Select Single

Returns the second argument if the first argument is greater than or equal to zero; returns the third argument otherwise.

Prototype

double __fsel (double, double, double);

float __fsels (float, float, float);

Square root functions

__frsqrte, __frsqrtes
Purpose

Floating Reciprocal Square Root Estimate, Floating Reciprocal Square Root Estimate Single

Prototype

double __frsqrte (double);

float __frsqrtes (float);

Usage

__frsqrtes is valid only when **-qarch** is set to target POWER5+ or later processors.

__fsqrt, __fsqrts
Purpose

Floating Square Root, Floating Square Root Single

Prototype

double __fsqrt (double);

float __fsqrts (float);

Software division functions

__swdiv, __swdivs
Purpose

Software Divide, Software Divide Single

Divides the first argument by the second argument and returns the result.

Prototype

> double __swdiv (double, double);

> float __swdivs (float, float);

__swdiv_nochk, __swdivs_nochk
Purpose

Software Divide No Check, Software Divide No Check Single

Divides the first argument by the second argument, without performing range checking, and returns the result.

Prototype

> double __swdiv_nochk (double *a*, double *b*);

> float __swdivs_nochk (float *a*, float *b*);

Parameters

a Must not equal infinity. When **-qstrict** is in effect, *a* must have an absolute value greater than 2^{-970} and less than infinity.

b Must not equal infinity, zero, or denormalized values. When **-qstrict** is in effect, *b* must have an absolute value greater than 2^{-1022} and less than 2^{1021}.

Return value

The result must not be equal to positive or negative infinity. When **-qstrict** in effect, the result must have an absolute value greater than 2^{-1021} and less than 2^{1023}.

Usage

This function can provide better performance than the normal divide operator or the `__swdiv` built-in function in situations where division is performed repeatedly in a loop and when arguments are within the permitted ranges.

Store functions

__stfiw
Purpose

Store Floating Point as Integer Word

Stores the contents of the loworder 32 bits of *value*, without conversion, into the word in storage addressed by *addr*.

Prototype

> void __stfiw (const int* *addr*, double *value*);

Synchronization and atomic built-in functions

Synchronization and atomic built-in functions are grouped into the following categories:

- "Check lock functions"
- "Clear lock functions" on page 425
- "Compare and swap functions" on page 426
- "Fetch functions" on page 427
- "Load functions" on page 429
- "Store functions" on page 429
- "Synchronization functions" on page 430

Check lock functions

__check_lock_mp, __check_lockd_mp
Purpose

Check Lock on Multiprocessor Systems, Check Lock Doubleword on Multiprocessor Systems

Conditionally updates a single word or doubleword variable atomically.

Prototype

unsigned int __check_lock_mp (const int* *addr*, int *old_value*, int *new_value*);

unsigned int __check_lockd_mp (const long* *addr*, long *old_value*, long *new_value*);

Parameters

addr
: The address of the variable to be updated. Must be aligned on a 4-byte boundary for a single word or on an 8-byte boundary for a doubleword.

old_value
: The old value to be checked against the current value in *addr*.

new_value
: The new value to be conditionally assigned to the variable in *addr*,

Return value

Returns false (0) if the value in *addr* was equal to *old_value* and has been set to the *new_value*. Returns true (1) if the value in *addr* was not equal to *old_value* and has been left unchanged.

Usage

__check_lockd_mp is valid only in 64-bit mode.

__check_lock_up, __check_lockd_up
Purpose

Check Lock on Uniprocessor Systems, Check Lock Doubleword on Uniprocessor Systems

Conditionally updates a single word or doubleword variable atomically.

Prototype

> unsigned int __check_lock_up (const int* *addr*, int *old_value*, int *new_value*);

> unsigned int __check_lockd_up (const long* *addr*, long *old_value*, long *new_value*);

Parameters

addr
> The address of the variable to be updated. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

old_value
> The old value to be checked against the current value in *addr*.

new_value
> The new value to be conditionally assigned to the variable in *addr*,

Return value

Returns false (0) if the value in *addr* was equal to *old_value* and has been set to the new value. Returns true (1) if the value in *addr* was not equal to *old_value* and has been left unchanged.

Usage

__check_lockd_up is valid only in 64-bit mode.

Clear lock functions

__clear_lock_mp, __clear_lockd_mp
Purpose

Clear Lock on Multiprocessor Systems, Clear Lock Doubleword on Multiprocessor Systems

Atomic store of the *value* into the variable at the address *addr*.

Prototype

> void __clear_lock_mp (const int* *addr*, int *value*);

> void __clear_lockd_mp (const long* *addr*, long *value*);

Parameters

addr
> The address of the variable to be updated. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

value
> The new value to be assigned to the variable in *addr*,

Usage

`__clear_lockd_mp` is only valid in 64-bit mode.

__clear_lock_up, __clear_lockd_up
Purpose

Clear Lock on Uniprocessor Systems, Clear Lock Doubleword on Uniprocessor Systems

Atomic store of the *value* into the variable at the address *addr*.

Prototype

> void __clear_lock_up (const int* *addr*, int *value*);

> void __clear_lockd_up (const long* *addr*, long *value*);

Parameters

addr
> The address of the variable to be updated. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

value
> The new value to be assigned to the variable in *addr*.

Usage

`__clear_lockd_up` is only valid in 64-bit mode.

Compare and swap functions

__compare_and_swap, __compare_and_swaplp
Purpose

Conditionally updates a single word or doubleword variable atomically.

Prototype

> int __compare_and_swap (volatile int* *addr*, int* *old_val_addr*, int *new_val*);

> int __compare_and_swaplp (volatile long* *addr*, long* *old_val_addr*, long *new_val*);

Parameters

addr
> The address of the variable to be copied. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

old_val_addr
> The memory location into which the value in *addr* is to be copied.

new_val
    The value to be conditionally assigned to the variable in *addr*,

Return value

Returns true (1) if the value in *addr* was equal to *old_value* and has been set to the new value. Returns false (0) if the value in *addr* was not equal to *old_value* and has been left unchanged. In either case, the contents of the memory location specified by *addr* are copied into the memory location specified by *old_val_addr*.

Usage

The __compare_and_swap function is useful when a single word value must be updated only if it has not been changed since it was last read. If you use __compare_and_swap as a locking primitive, insert a call to the __isync built-in function at the start of any critical sections.

__compare_and_swaplp is valid only in 64-bit mode.

Fetch functions

__fetch_and_and, __fetch_and_andlp
Purpose

Clears bits in the word or doubleword specified by*addr* by AND-ing that value with the value specified by *val*, in a single atomic operation, and returns the original value of *addr*.

Prototype

> unsigned int __fetch_and_and (volatile unsigned int* *addr*, unsigned int *val*);

> unsigned long __fetch_and_andlp (volatile unsigned long* *addr*, unsigned long *val*);

Parameters

addr
    The address of the variable to be ANDed. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

value
    The value by which the value in *addr* is to be ANDed.

Usage

This operation is useful when a variable containing bit flags is shared between several threads or processes.

__fetch_and_andlp is valid only in 64-bit mode.

__fetch_and_or, __fetch_and_orlp
Purpose

Sets bits in the word or doubleword specified by *addr* by OR-ing that value with the value specified *val*, in a single atomic operation, and returns the original value of *addr*.

Prototype

unsigned int __fetch_and_or (volatile unsigned int* *addr*, unsigned int *val*);

unsigned long __fetch_and_orlp (volatile unsigned long* *addr*, unsigned long *val*);

Parameters

addr
  The address of the variable to be ORed. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

value
  The value by which the value in *addr* is to be ORed.

Usage

This operation is useful when a variable containing bit flags is shared between several threads or processes.

__fetch_and_orlp is valid only in 64-bit mode.

__fetch_and_swap, __fetch_and_swaplp
Purpose

Sets the word or doubleword specified by *addr* to the value of *val* and returns the original value of *addr*, in a single atomic operation.

Prototype

unsigned int __fetch_and_swap (volatile unsigned int* *addr*, unsigned int *val*);

unsigned long __fetch_and_swaplp (volatile unsigned long* *addr*, unsigned long *val*);

Parameters

addr
  The address of the variable to be updated. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

value
  The value which is to be assigned to *addr*.

Usage

This operation is useful when a variable is shared between several threads or processes, and one thread needs to update the value of the variable without losing the value that was originally stored in the location.

__fetch_and_swaplp is valid only in 64-bit mode.

Load functions

__ldarx, __lwarx
Purpose

Load Doubleword and Reserve Indexed, Load Word and Reserve Indexed

Loads the value from the memory location specified by *addr* and returns the result. For __lwarx, in 64-bit mode, the compiler returns the sign-extended result.

Prototype

long __ldarx (volatile long* *addr*);

int __lwarx (volatile int* *addr*);

Parameters

addr
> The address of the value to be loaded. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

Usage

This function can be used with a subsequent __stdcx (or __stwcx) built-in function to implement a read-modify-write on a specified memory location. The two built-in functions work together to ensure that if the store is successfully performed, no other processor or mechanism can modify the target doubleword between the time the __ldarx function is executed and the time the __stdcx function completes. This has the same effect as inserting __fence built-in functions before and after the __ldarx built-in function and can inhibit compiler optimization of surrounding code (see "__alignx" on page 501 for a description of the __fence built-in function).

__ldarx is valid only in 64-bit mode.

Store functions

__stdcx, __stwcx
Purpose

Store Doubleword Conditional Indexed, Store Word Conditional Indexed

Stores the value specified by *val* into the memory location specified by *addr*.

Prototype

int __stdcx(volatile long* *addr*, long *val*);

int __stwcx(volatile int* *addr*, int *val*);

Parameters

addr
> The address of the variable to be updated. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

value
>   The value which is to be assigned to *addr*.

Return value

Returns 1 if the update of *addr* is successful and 0 if it is unsuccessful.

Usage

This function can be used with a preceding __ldarx (or __lwarx) built-in function to implement a read-modify-write on a specified memory location. The two built-in functions work together to ensure that if the store is successfully performed, no other processor or mechanism can modify the target doubleword between the time the __ldarx function is executed and the time the __stdcx function completes. This has the same effect as inserting __fence built-in functions before and after the __stdcx built-in function and can inhibit compiler optimization of surrounding code.

__stdcx is valid only in 64-bit mode.

Synchronization functions

__eieio, __iospace_eioio
Purpose

Enforce In-order Execution of Input/Output

Ensures that all I/O storage access instructions preceding the call to __eioeio complete in main memory before I/O storage access instructions following the function call can execute.

Prototype

>   void __eieio (void);

>   void __iospace_eieio (void);

Usage

This function is useful for managing shared data instructions where the execution order of load/store access is significant. The function can provide the necessary functionality for controlling I/O stores without the cost to performance that can occur with other synchronization instructions.

__isync
Purpose

Instruction Synchronize

Waits for all previous instructions to complete and then discards any prefetched instructions, causing subsequent instructions to be fetched (or refetched) and executed in the context established by previous instructions.

Prototype

>   void __isync (void);

__lwsync, __iospace_lwsync
Purpose

Load Word Synchronize

Ensures that all instructions preceding the call to __lwsync complete before any subsequent store instructions can be executed on the processor that executed the function. Also, it ensures that all load instructions preceding the call to __lwsync complete before any subsequent load instructions can be executed on the processor that executed the function. This allows you to synchronize between multiple processors with minimal performance impact, as __lwsync does not wait for confirmation from each processor.

Prototype

> void __lwsync (void);

> void __iospace_lwsync (void);

__sync, __iospace_sync
Purpose

Synchronize

Ensures that all instructions preceding the function the call to __sync complete before any instructions following the function call can execute.

Prototype

> void __sync (void);

> void __iospace_sync (void);

Cache-related built-in functions

Cache-related built-in functions are grouped into the following categories:
- "Data cache functions"
- "Prefetch built-in functions" on page 433

Data cache functions

__dcbf
Purpose

Data Cache Block Flush

Copies the contents of a modified block from the data cache to main memory and flushes the copy from the data cache.

Prototype

> void __dcbf(const void* *addr*);

__dcbfl
Purpose

Data Cache Block Flush Line

Flushes the cache line at the specified address from the L1 data cache.

Prototype

void __dcbfl (const void* *addr*);

Usage

The target storage block is preserved in the L2 cache.

Valid only when **-qarch** is set to target POWER6 processors

__dcbst
Purpose

Data Cache Block Store

Copies the contents of a modified block from the data cache to main memory.

Prototype

void __dcbst(const void* *addr*);

__dcbt
Purpose

Data Cache Block Touch

Loads the block of memory containing the specified address into the L1 data cache.

Prototype

void __dcbt (void* *addr*);

__dcbtst
Purpose

Data Cache Block Touch for Store

Fetches the block of memory containing the specified address into the data cache.

Prototype

void __dcbtst(void* *addr*);

__dcbz
Purpose

Data Cache Block set to Zero

Sets a cache line containing the specified address in the data cache to zero (0).

Prototype

void __dcbz (void* *addr*);

__dcbflp
Purpose

Data Cache Block Flush Line Primary

Flushes the cache line at address from the primary data cache of a single processor.

Prototype

void __dcbflp(const void* address);

Usage

Valid only when **-qarch** is set to target POWER7 processors.

Prefetch built-in functions

__prefetch_by_load
Purpose

Touches a memory location by using an explicit load.

Prototype

void __prefetch_by_load (const void*);

__prefetch_by_stream
Purpose

Touches consecutive memory locations by using an explicit stream.

Prototype

void __prefetch_by_stream (const int, const void*);

__dcbtstt
Purpose

Store Transient Touch provides a hint that describes a block that the program may perform a store access to. The block is likely to be transient, that is, the time interval during which the program accesses the unit is likely to be short.

Prototype

void __dcbtstt (void * address);

Usage

Valid only when **-qarch** is set to target POWER7 processors.

__dcbtt
Purpose

Data Cache Block Touch Transient

Load Transient Touch provides a hint that describes a block that the program might perform a load access to. The block is likely to be transient, that is, the time interval during which the program accesses the unit is likely to be short.

Prototype

    void __dcbtt (void * address);

Usage

Valid only when **-qarch** is set to target POWER7 processors.

__partial_dcbt
Purpose

Partial Data Cache Block Touch

Loads half of the cache line that contains the specified address into the L3 data cache.

Prototype

    void __partial_dcbt (void * address);

Usage

Valid only when **-qarch** is set to target POWER7 processors.

__protected_stream_stride
Purpose

Sets the word-offset of the first unit of the stream address_offset, and stride in word size for protected load or store stream with identifier stream_id

Prototype

    void __protected_stream_stride (unsigned int *address_offset*, unsigned int *stride*, unsigned int *stream_id*);

Parameters

address_offset
> The address of the first unit of the prefetch variable.

stride
> This is the distance in the number of words of two consecutive elements of the prefetch stream.

stream_id
> An integer with a value 0 to 11 on POWER7 processors.

Usage

Valid only when -qarch is set to target POWER7 processors.

__transient_protected_stream_count_depth
Purpose

Sets the number of cache lines *unit_cnt* and the prefetch depth *prefetch_depth* for the limited length protected load or store stream with identifier *stream_id*. The term "transient" indicates that the time interval during which the program accesses the stream's memory is likely to be short, so the processor can remove it from the cache earlier.

Prototype

> void __transient_protected_stream_count_depth (unsigned int *unit_cnt*, unsigned int *prefetch_depth*, unsigned int *stream_id*);

Parameters

unit_cnt
> The number of cache lines. Must be an integer with a value of 0 to 1023.

prefetch_depth
> A relative, qualitative value which sets the steady-state *fetch-ahead* distance of the prefetches for a stream. The fetch-ahead distance is the number of lines being prefetched in advance of the line from which data is currently being loaded, or to which data is currently being stored. Valid values are as follows:

> **0** The default defined in the Data Stream Control Register.

> **1** None.

> **2** Shallowest.

> **3** Shallow.

> **4** Medium.

> **5** Deep.

> **6** Deeper.

> **7** Deepest.

stream_id
> An integer with a value 0 to 11 on POWER7 processors.

Usage

Valid only when **-qarch** is set to target POWER7 processors.

__transient_unlimited_protected_stream_depth
Purpose

Sets the prefetch depth prefetch_depth for the unlimited length protected load or store stream with identifier *stream_id*. The stream is likely to be transient, that is, the time interval during which the program accesses the unit is likely to be short.

Prototype

> void __transient_unlimited_protected_stream_depth (unsigned int *prefetch_depth*, unsigned int *stream_id*);

Parameters

prefetch_depth
> A relative, qualitative value which sets the steady-state *fetch-ahead* distance of the prefetches for a stream. The fetch-ahead distance is the number of lines being prefetched in advance of the line from which data is currently being loaded, or to which data is currently being stored. Valid values are as follows:

> **0**   The default defined in the Data Stream Control Register.

> **1**   None.

> **2**   Shallowest.

> **3**   Shallow.

> **4**   Medium.

> **5**   Deep.

> **6**   Deeper.

> **7**   Deepest.

stream_id
> An integer with a value 0 to 11 on POWER7 processors.

Usage

Valid only when **-qarch** is set to target POWER7 processors.

__unlimited_protected_stream_depth
Purpose

Sets the prefetch depth *prefetch_depth* for the unlimited length protected load or store stream with identifier *stream_id*.

Prototype

> void __unlimited_protected_stream_depth (unsigned in *prefetch_depth*, unsigned int *stream_id*);

Parameter

prefetch_depth
> A relative, qualitative value which sets the steady-state *fetch-ahead* distance of the prefetches for a stream. The fetch-ahead distance is the number of lines being prefetched in advance of the line from which data is currently being loaded, or to which data is currently being stored. Valid values are as follows:

> **0**   The default defined in the Data Stream Control Register.

> **1**   None.

> **2**   Shallowest.

> **3**   Shallow.

> **4**   Medium.

5 Deep.

6 Deeper.

7 Deepest.

stream_id
> An integer with a value 0 to 15 on POWER6 processors, and a value 0 to 11 on POWER7 processors.

Usage

Valid only when **-qarch** is set to target POWER6 or POWER7 processors.

__protected_store_stream_set, __protected_unlimited_store_stream_set
Purpose

Establishes a limited- or unlimited-length protected store stream which fetches from either incremental (forward) or decremental (backward) memory addresses. The stream is protected from being replaced by any hardware detected streams.

Prototype

> void _protected_store_stream_set (unsigned int *direction*, const void* *addr*, unsigned int *stream_ID*);

> void _protected_unlimited_store_stream_set (unsigned int *direction*, const void* *addr*, unsigned int *stream_ID*);

Parameters

direction
> An integer with a value of 1 (forward) or 3 (backward).

addr
> The beginning of the cache line.

stream_ID
> An integer with a value 0 to 15 on POWER6 processors, and a value 0 to 11 on POWER7 processors.

Usage

Valid only when **-qarch** is set to target POWER6 and POWER7 processors.

__protected_stream_count
Purpose

Sets the number of cache lines for a specific limited-length protected stream.

Prototype

> void __protected_stream_count (unsigned int *unit_cnt*, unsigned int *stream_ID*);

Parameters

unit_cnt
> The number of cache lines. Must be an integer with a value of 0 to 1023.

stream_ID
> An integer with a value 0-7 on POWER5 processors, a value 0 to 15 on
> POWER6 processors, and a value 0 to 11 on POWER7 processors.

Usage

Valid only when **-qarch** is set to target POWER5, POWER6, or POWER7
processors.

__protected_stream_count_depth
Purpose

Sets the number of cache lines and the prefetch depth for a specific limited-length
protected stream.

Prototype

> void _protected_stream_count_depth (unsigned int *unit_cnt*, unsigned int
> *prefetch_depth*, unsigned int *stream_ ID*);

Parameters

unit_cnt
> The number of cache lines. Must be an integer with a value of 0 to 1023.

prefetch_depth
> A relative, qualitative value which sets the steady-state *fetch-ahead* distance of
> the prefetches for a stream. The fetch-ahead distance is the number of lines
> being prefetched in advance of the line from which data is currently being
> loaded, or to which data is currently being stored. Valid values are as follows:

> **0** The default defined in the Data Stream Control Register.

> **1** None.

> **2** Shallowest.

> **3** Shallow.

> **4** Medium.

> **5** Deep.

> **6** Deeper.

> **7** Deepest.

stream_ID
> An integer with a value 0 to 15 on POWER6 processors, and a value 0 to 11 on
> or POWER7 processors.

Usage

Valid only when **-qarch** is set to target POWER6 or POWER7 processors.

__protected_stream_go
Purpose

Starts prefetching all limited-length protected streams.

Prototype

> void __protected_stream_go (void);

Usage

Valid only when **-qarch** is set to target POWER5, POWER6, or POWER7 processors.

__protected_stream_set, __protected_unlimited_stream_set, __protected_unlimited_stream_set_go
Purpose

Establishes a limited- or unlimited-length protected stream which fetches from either incremental (forward) or decremental (backward) memory addresses. The stream is protected from being replaced by any hardware detected streams.

Prototype

> void __protected_stream_set (unsigned int *direction*, const void* *addr*, unsigned int *stream_ID*);
>
> void _protected_unlimited_stream_set (unsigned int *direction*, const void* *addr*, unsigned int *ID*);
>
> void __protected_unlimited_stream_set_go (unsigned int *direction*, const void* *addr*, unsigned int *stream_ID*);

Parameters

direction
    An integer with a value of 1 (forward) or 3 (backward).

addr
    The beginning of the cache line.

stream_ID
    An integer with a value 0-7 on POWER4 and POWER5 processors, a value 0 to 15 on POWER6 processors, and a value 0 to 11 on POWER7 processors.

Usage

Valid only when **-qarch** is set to target POWER5, POWER6, or POWER7 processors.

__protected_stream_stop
Purpose

Stops prefetching a protected stream.

Prototype

> void __protected_stream_stop (unsigned int *stream ID*);

Parameters

stream_id
> An integer with a value 0-7 on POWER5 processors, a value 0 to 15 on POWER6 processors, and a value 0 to 11 on POWER7 processors.

Usage

Valid only when **-qarch** is set to target POWER5, POWER6, or POWER7 processors.

__protected_stream_stop_all
Purpose

Stops prefetching all protected streams.

Prototype

> void __protected_stream_stop_all (void);

Usage

Valid only when **-qarch** is set to target POWER5, POWER6, or POWER7 processors.

Block-related built-in functions

__bcopy
Purpose

Copies *n* bytes from *src* to *dest*. The result is correct even when both areas overlap.

Prototype

> void __bcopy(const void* *src*, void* *dest*, size_t *n*);

Parameters

src The source address of data to be copied.

dest
> The destination address of data to be copied

n The size of the data.

Vector built-in functions

Individual elements of vectors can be accessed by using the Vector Multimedia eXtension (VMX) or the Vector Scalar eXtension (VSX) built-in functions. This section provides an alphabetical reference to the VMX and the VSX built-in functions. You can use these functions to manipulate vectors.

You must specify appropriate compiler options for your architecture when you use the built-in functions. Built-in functions that use or return a **vector unsigned long long**, **vector signed long long**, **vector bool long long**, or **vector double** type

require an architecture that supports the VSX instruction set extensions, such as POWER7. You must specify **-qarch=pwr7** when you use these types.

This section uses pseudocode description to represent function syntax, as shown below:

```
d=func_name(a, b, c)
```

In the description,
- d represents the return value of the function.
- a, b, and c represent the arguments of the function.
- func_name is the name of the function.

For example, the syntax for the function `vector double vec_xld2(int, double*);` is represented by d=vec_xld2(a, b).

Note: This section only describes the IBM specific vector built-in functions and the AltiVec built-in functions with IBM extensions. For information about the other AltiVec built-in functions, see the AltiVec Application Programming Interface specification.

vec_abs
Purpose

Returns a vector containing the absolute values of the contents of the given vector.

Syntax

```
d=vec_abs(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector signed char	vector signed char
vector signed short	vector signed short
vector signed int	vector signed int
vector float	vector float
vector double	vector double

Result value

The value of each element of the result is the absolute value of the corresponding element of a.

vec_add
Purpose

Returns a vector containing the sums of each set of corresponding elements of the given vectors.

This function emulates the operation on long long vectors.

Syntax

```
d=vec_add(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector float	vector float
vector double	vector double	vector double

Result value

The value of each element of the result is the sum of the corresponding elements of a and b. For integer vectors and unsigned vectors, the arithmetic is modular.

vec_all_eq

Purpose

Tests whether all sets of corresponding elements of the given vectors are equal.

Syntax

```
d=vec_all_eq(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
int	vector bool char	vector bool char
		vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector bool short
		vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector bool int
		vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector bool long long
		vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if each element of a is equal to the corresponding element of b. Otherwise, the result is 0.

vec_all_ge
Purpose

Tests whether all elements of the first argument are greater than or equal to the corresponding elements of the second argument.

Syntax

```
d=vec_all_ge(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if all elements of a are greater than or equal to the corresponding elements of b. Otherwise, the result is 0.

vec_all_gt
Purpose

Tests whether all elements of the first argument are greater than the corresponding elements of the second argument.

Syntax

```
d=vec_all_gt(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if all elements of a are greater than the corresponding elements of b. Otherwise, the result is 0.

vec_all_le
Purpose

Tests whether all elements of the first argument are less than or equal to the corresponding elements of the second argument.

Syntax

```
d=vec_all_le(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if all elements of a are less than or equal to the corresponding elements of b. Otherwise, the result is 0.

vec_all_lt
Purpose

Tests whether all elements of the first argument are less than the corresponding elements of the second argument.

Syntax

```
d=vec_all_lt(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if all elements of a are less than the corresponding elements of b. Otherwise, the result is 0.

vec_all_nan
Purpose

Tests whether each element of the given vector is a NaN.

Syntax

```
d=vec_all_nan(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
int	vector float
	vector double

Result value

The result is 1 if each element of a is a NaN. Otherwise, the result is 0.

vec_all_ne
Purpose

Tests whether all sets of corresponding elements of the given vectors are not equal.

Syntax

```
d=vec_all_ne(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if each element of a is not equal to the corresponding element of b. Otherwise, the result is 0.

vec_all_nge
Purpose

Tests whether each element of the first argument is not greater than or equal to the corresponding element of the second argument.

Syntax

```
d=vec_all_nge(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
int	vector float	vector float
	vector double	vector double

Result value

The result is 1 if each element of a is not greater than or equal to the corresponding element of b. Otherwise, the result is 0.

vec_all_ngt
Purpose

Tests whether each element of the first argument is not greater than the corresponding element of the second argument.

Syntax

```
d=vec_all_ngt(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
int	vector float	vector float
	vector double	vector double

Result value

The result is 1 if each element of a is not greater than the corresponding element of b. Otherwise, the result is 0.

vec_all_nle
Purpose

Tests whether each element of the first argument is not less than or equal to the corresponding element of the second argument.

Syntax

```
d=vec_all_nle(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
int	vector float	vector float
	vector double	vector double

Result value

The result is 1 if each element of a is not less than or equal to the corresponding element of b. Otherwise, the result is 0.

vec_all_nlt
Purpose

Tests whether each element of the first argument is not less than the corresponding element of the second argument.

Syntax

```
d=vec_all_nlt(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
int	vector float	vector float
	vector double	vector double

Result value

The result is 1 if each element of a is not less than the corresponding element of b. Otherwise, the result is 0.

vec_all_numeric
Purpose

Tests whether each element of the given vector is numeric (not a NaN).

Syntax

```
d=vec_all_numeric(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
int	vector float
	vector double

Result value

The result is 1 if each element of a is numeric (not a NaN). Otherwise, the result is 0.

vec_and

Purpose

Performs a bitwise AND of the given vectors.

Syntax

`d=vec_and(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector bool char	vector bool char	vector bool char
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector bool short	vector bool short	vector vector bool short
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector bool int	vector bool int	vector bool int
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector bool long long	vector signed long long
	vector signed long long	vector signed long long
		vector bool long long

d	a	b
vector unsigned long long	vector bool long long	vector unsigned long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
vector float	vector bool int	vector float
	vector float	vector bool int
		vector float
vector double	vector bool long long	vector double
	vector double	vector double
		vector bool long long

vec_andc

Purpose

Performs a bitwise AND of the first argument and the bitwise complement of the second argument.

Syntax

```
d=vec_andc(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector bool char	vector bool char	vector bool char
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector bool short	vector bool short	vector bool short
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector bool int	vector bool int	vector bool int
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int

d	a	b
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector bool long long	vector signed long long
	vector signed long long	vector signed long long
		vector bool long long
vector unsigned long long	vector bool long long	vector unsigned long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
vector float	vector bool int	vector float
	vector float	vector bool int
		vector float
vector double	vector bool long long	vector double
	vector double	vector bool long long
		vector double

Result value

The result is the bitwise AND of a with the bitwise complement of b.

vec_any_eq
Purpose

Tests whether any set of corresponding elements of the given vectors are equal.

Syntax

```
d=vec_any_eq(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
int	vector bool char	vector bool char
		vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector bool short
		vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector bool int
		vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector bool long long
		vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of a is equal to the corresponding element of b. Otherwise, the result is 0.

vec_any_ge
Purpose

Tests whether any element of the first argument is greater than or equal to the corresponding element of the second argument.

Syntax

```
d=vec_any_ge(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of a is greater than or equal to the corresponding element of b. Otherwise, the result is 0.

vec_any_gt
Purpose

Tests whether any element of the first argument is greater than the corresponding element of the second argument.

Syntax

```
d=vec_any_gt(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of a is greater than the corresponding element of b. Otherwise, the result is 0.

vec_any_le
Purpose

Tests whether any element of the first argument is less than or equal to the corresponding element of the second argument.

Syntax

```
d=vec_any_le(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of a is less than or equal to the corresponding element of b. Otherwise, the result is 0.

vec_any_lt
Purpose

Tests whether any element of the first argument is less than the corresponding element of the second argument.

Syntax

```
d=vec_any_lt(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of a is less than the corresponding element of b. Otherwise, the result is 0.

vec_any_nan
Purpose

Tests whether any element of the given vector is a NaN.

Syntax

```
d=vec_any_nan(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
int	vector float
	vector double

Result value

The result is 1 if any element of a is a NaN. Otherwise, the result is 0.

vec_any_ne
Purpose

Tests whether any set of corresponding elements of the given vectors are not equal.

Syntax

```
d=vec_any_ne(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
int	vector bool char	vector bool char
		vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector bool short
		vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector bool int
		vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector bool long long
		vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of a is not equal to the corresponding element of b. Otherwise, the result is 0.

vec_any_nge
Purpose

Tests whether any element of the first argument is not greater than or equal to the corresponding element of the second argument.

Syntax

```
d=vec_any_nge(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
int	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of a is not greater than or equal to the corresponding element of b. Otherwise, the result is 0.

vec_any_ngt
Purpose

Tests whether any element of the first argument is not greater than the corresponding element of the second argument.

Syntax

```
d=vec_any_ngt(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
int	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of a is not greater than the corresponding element of b. Otherwise, the result is 0.

vec_any_nle
Purpose

Tests whether any element of the first argument is not less than or equal to the corresponding element of the second argument.

Syntax

```
d=vec_any_nle(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
int	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of a is not less than or equal to the corresponding element of b. Otherwise, the result is 0.

vec_any_nlt
Purpose

Tests whether any element of the first argument is not less than the corresponding element of the second argument.

Syntax

```
d=vec_any_nlt(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
int	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of a is not less than the corresponding element of b. Otherwise, the result is 0.

vec_any_numeric
Purpose

Tests whether any element of the given vector is numeric (not a NaN).

Syntax

```
d=vec_any_numeric(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
int	vector float
	vector double

Result value

The result is 1 if any element of a is numeric (not a NaN). Otherwise, the result is 0.

vec_ceil
Purpose

Returns a vector containing the smallest representable floating-point integer values greater than or equal to the values of the corresponding elements of the given vector.

Note: vec_ceil is another name for vec_roundp. For details, see "vec_roundp" on page 486.

vec_cmpeq
Purpose

Returns a vector containing the results of comparing each set of corresponding elements of the given vectors for equality.

This function emulates the operation on long long vectors.

Syntax

d=vec_cmpeq(a, b)

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector bool char	vector bool char	vector bool char
	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
vector bool short	vector bool short	vector bool short
	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
vector bool int	vector bool int	vector bool int
	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
	vector float	vector float

d	a	b
vector bool long long	vector bool long long	vector bool long long
	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long
	vector double	vector double

Result value

For each element of the result, the value of each bit is 1 if the corresponding elements of a and b are equal. Otherwise, the value of each bit is 0.

vec_cmpge
Purpose

Returns a vector containing the results of a greater-than-or-equal-to comparison between each set of corresponding elements of the given vectors.

Syntax

```
d=vec_cmpge(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector bool char	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
vector bool short	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
vector bool int	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
	vector float	vector float
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long
	vector double	vector double

Result value

For each element of the result, the value of each bit is 1 if the value of the corresponding element of a is greater than or equal to the value of the corresponding element of b. Otherwise, the value of each bit is 0.

vec_cmpgt
Purpose

Returns a vector containing the results of a greater-than comparison between each set of corresponding elements of the given vectors.

This function emulates the operation on long long vectors.

Syntax

```
d=vec_cmpgt(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector bool char	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
vector bool short	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
vector bool int	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
	vector float	vector float
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long
	vector double	vector double

Result value

For each element of the result, the value of each bit is 1 if the value of the corresponding element of a is greater than the value of the corresponding element of b. Otherwise, the value of each bit is 0.

vec_cmple
Purpose

Returns a vector containing the results of a less-than-or-equal-to comparison between each set of corresponding elements of the given vectors.

Syntax

```
d=vec_cmple(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector bool char	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
vector bool short	vector signed short	vector signed short
	vector unsigned short	vector unsigned short

d	a	b
vector bool int	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
	vector float	vector float
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long
	vector double	vector double

Result value

For each element of the result, the value of each bit is 1 if the value of the corresponding element of a is less than or equal to the value of the corresponding element of b. Otherwise, the value of each bit is 0.

vec_cmplt

Purpose

Returns a vector containing the results of a less-than comparison between each set of corresponding elements of the given vectors.

This operation emulates the operation on long long vectors.

Syntax

```
d=vec_cmplt(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector bool char	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
vector bool short	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
vector bool int	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
	vector float	vector float
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long
	vector double	vector double

Result value

For each element of the result, the value of each bit is 1 if the value of the corresponding element of a is less than the value of the corresponding element of b. Otherwise, the value of each bit is 0.

vec_cpsgn

Purpose

Returns a vector by copying the sign of the elements in vector a to the sign of the corresponding elements in vector b.

This function requires the POWER7 architecture.

Syntax

```
d=vec_cpsgn(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector float	vector float	vector float
vector double	vector double	vector double

vec_ctd

Purpose

Converts the type of each element in a from integer to floating-point single precision and divides the result by 2 to the power of b.

Syntax

```
d=vec_ctd(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector double	vector signed int	0–31
	vector unsigned int	
	vector signed long long	
	vector unsigned long long	

vec_ctf

Purpose

Converts a vector of fixed-point numbers into a vector of floating-point numbers.

Syntax

```
d=vec_ctf(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector float	vector signed int	0–31
	vector unsigned int	
	vector signed long long	
	vector unsigned long long	

Result value

The value of each element of the result is the closest floating-point estimate of the value of the corresponding element of a divided by 2 to the power of b.

vec_cts
Purpose

Converts a vector of floating-point numbers into a vector of signed fixed-point numbers.

Syntax

```
d=vec_cts(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector signed int	vector float	0–31
	vector double	

Result value

The value of each element of the result is the saturated value obtained by multiplying the corresponding element of a by 2 to the power of b.

vec_ctsl
Purpose

Multiplies each element in a by 2 to the power of b and converts the result into an integer.

Note: This function does not use elements 1 and 3 of a when a is a double vector.

Syntax

```
d=vec_ctsl(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector signed long long	vector float	0–31
	vector double	

vec_ctu
Purpose

Converts a vector of floating-point numbers into a vector of unsigned fixed-point numbers.

Note: Elements 1 and 3 of the result vector are undefined when a is a double vector.

Syntax

```
d=vec_ctu(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector unsigned int	vector float	0–31
	vector double	

Result value

The value of each element of the result is the saturated value obtained by multiplying the corresponding element of a by 2 to the power of b.

vec_ctul
Purpose

Multiplies each element in a by 2 to the power of b and converts the result into an unsigned type.

Syntax

```
d=vec_ctul(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector unsigned long long	vector float	0–31
	vector double	

Result value

This function does not use elements 1 and 3 of a when a is a float vector.

vec_cvf

Purpose

Converts a single-precision floating-point vector to a double-precision floating-point vector or converts a double-precision floating-point vector to a single-precision floating-point vector.

Syntax

```
d=vec_cvf(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector float	vector double
vector double	vector float

Result value

When this function converts from vector float to vector double, it converts the types of elements 0 and 2 in the vector.

When this function converts from vector double to vector float, the types of element 1 and 3 in the result vector are undefined.

vec_div

Purpose

Divides the elements in vector a by the corresponding elements in vector b and then assigns the result to corresponding elements in the result vector.

This function emulates the operation on integer vectors. It requires the POWER7 architecture.

Syntax

```
d=vec_div(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector float	vector float
vector double	vector double	vector double

vec_extract
Purpose

Returns the value of element b from the vector a.

Syntax

```
d=vec_extract(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
signed char	vector signed char	signed int
unsigned char	vector unsigned char	
	vector bool char	
signed short	vector signed short	
unsigned short	vector unsigned short	
	vector bool short	
signed int	vector signed int	
unsigned int	vector unsigned int	
	vector bool int	
signed long long	vector signed long long	
unsigned long long	vector unsigned long long	
	vector bool long long	
float	vector float	
double	vector double	

Result value

This function uses the modulo arithmetic on b to determine the element number. For example, if b is out of range, the compiler uses b modulo the number of

elements in the vector to determine the element position.

vec_floor

Purpose

Returns a vector containing the largest representable floating-point integer values less than or equal to the values of the corresponding elements of the given vector.

Note: `vec_floor` is another name for `vec_roundm`. For details, see "vec_roundm" on page 486.

vec_insert

Purpose

Returns a copy of the vector b with the value of its element c replaced by a.

Syntax

`d=vec_insert(a, b, c)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector signed char	signed char	vector signed char	signed int
vector unsigned char	unsigned char	vector bool char	
		vector unsigned char	
vector signed short	signed short	vector signed short	
vector unsigned short	unsigned short	vector bool short	
		vector unsigned short	
vector signed int	signed int	vector signed int	
vector unsigned int	unsigned int	vector bool int	
		vector unsigned int	
vector signed long long	signed long long	vector signed long long	
vector unsigned long long	unsigned long long	vector bool long long	
		vector unsigned long long	
vector float	float	vector float	
vector double	double	vector double	

Result value

This function uses the modulo arithmetic on c to determine the element number. For example, if c is out of range, the compiler uses c modulo the number of elements in the vector to determine the element position.

vec_madd

Purpose

Returns a vector containing the results of performing a fused multiply/add for each corresponding set of elements of the given vectors.

Syntax

`d=vec_madd(a, b, c)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector float	vector float	vector float	vector float
vector double	vector double	vector double	vector double

Result value

The value of each element of the result is the product of the values of the corresponding elements of a and b, added to the value of the corresponding element of c.

vec_max

Purpose

Returns a vector containing the maximum value from each set of corresponding elements of the given vectors.

Syntax

`d=vec_max(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short

d	a	b
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector float	vector float	vector float
vector double	vector double	vector double

Result value

The value of each element of the result is the maximum of the values of the corresponding elements of a and b.

vec_mergeh
Purpose

Merges the most significant halves of two vectors.

Syntax

```
d=vec_mergeh(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector bool char	vector bool char	vector bool char
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector bool short	vector bool short	vector bool short
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector bool int	vector bool int	vector bool int
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector float	vector float
vector double	vector double	vector double

Result value

Assume that the elements of each vector are numbered beginning with 0. The even-numbered elements of the result are taken, in order, from the elements in the most significant 8 bytes of a. The odd-numbered elements of the result are taken, in order, from the elements in the most significant 8 bytes of b.

vec_mergel
Purpose

Merges the least significant halves of two vectors.

Syntax

```
d=vec_mergel(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector bool char	vector bool char	vector bool char
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector bool short	vector bool short	vector bool short
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector bool int	vector bool int	vector bool int
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector float	vector float
vector double	vector double	vector double

Result value

Assume that the elements of each vector are numbered beginning with 0. The even-numbered elements of the result are taken, in order, from the elements in the least significant 8 bytes of a. The odd-numbered elements of the result are taken, in order, from the elements in the least significant 8 bytes of b.

vec_min
Purpose

Returns a vector containing the minimum value from each set of corresponding elements of the given vectors.

Syntax

```
d=vec_min(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector float	vector float	vector float
vector double	vector double	vector double

Result value

The value of each element of the result is the minimum of the values of the corresponding elements of a and b.

vec_msub
Purpose

Returns a vector containing the results of performing a multiply-substract operation using the given vectors.

This function requires the POWER7 architecture.

Syntax

```
d=vec_msub(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector float	vector float	vector float	vector float
vector double	vector double	vector double	vector double

Result value

This function multiplies each element in a by the corresponding element in b and then substracts the corresponding element in c from the result.

vec_mul
Purpose

Returns a vector containing the results of performing a multiply operation using the given vectors.

This function emulates the operation on integer vectors. It requires the POWER7 architecture.

Syntax

```
d=vec_mul(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector float	vector float
vector double	vector double	vector double

Result value

This function multiplies corresponding elements in the given vectors and then assigns the result to corresponding elements in the result vector.

vec_nabs

Purpose

Returns a vector containing the results of performing a negative-absolute operation using the given vector.

This function requires the POWER7 architecture.

Syntax

```
d=vec_nabs(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector float	vector float
vector double	vector double

Result value

This function computes the absolute value of each element in the given vector and then assigns the negative value of the result to the corresponding elements in the result vector.

vec_neg

Purpose

Returns a vector containing the negative value of the corresponding elements in the given vector.

Note: For `vector signed long long`, this function emulates the operation. This function requires the POWER7 architecture.

Syntax

```
d=vec_neg(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector signed char	vector signed char
vector signed short	vector signed short
vector signed int	vector signed int
vector signed long long	vector signed long long
vector float	vector float
vector double	vector double

Result value

This function multiplies the value of each element in the given vector by -1 and then assigns the result to the corresponding elements in the result vector.

vec_nmadd
Purpose

Returns a vector containing the results of performing a negative multiply-sum operation on the given vectors.

This function requires the POWER7 architecture.

Syntax

```
d=vec_nmadd(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector float	vector float	vector float	vector float
vector double	vector double	vector double	vector double

Result value

The value of each element of the result is the product of the corresponding elements of a and b, added by the corresponding elements of c, and then multiplied by -1.0.

vec_nmsub
Purpose

Returns a vector containing the results of performing a negative multiply-subtract operation on the given vectors.

Syntax

```
d=vec_nmsub(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector float	vector float	vector float	vector float
vector double	vector double	vector double	vector double

Result value

The value of each element of the result is the product of the corresponding elements of a and b, subtracted from the corresponding element of c.

vec_nor
Purpose

Performs a bitwise NOR of the given vectors.

Syntax

```
d=vec_nor(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector bool char	vector bool char	vector bool char
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector bool short	vector bool short	vector vector bool short
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector bool int	vector bool int	vector bool int
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector bool int	vector float
	vector float	vector bool int
vector double	vector double	vector double

Result value

The result is the bitwise NOR of a and b.

vec_or

Purpose

Performs a bitwise OR of the given vectors.

Syntax

```
d=vec_or(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector bool char	vector bool char	vector bool char
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector bool short	vector bool short	vector vector bool short
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector bool int	vector bool int	vector bool int
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector bool long long	vector signed long long
	vector signed long long	vector signed long long
		vector bool long long
vector unsigned long long	vector bool long long	vector unsigned long long
	vector unsigned long long	vector unsigned long long
		vector bool long long

d	a	b
vector float	vector bool int	vector float
	vector float	vector bool int
		vector float
vector double	vector bool long long	vector double
	vector double	vector bool long long
		vector double

Result value

The result is the bitwise OR of a and b.

vec_permi
Purpose

Returns a vector by permuting and combining the two eight-byte-long vector elements in a and b based on the value of c.

Syntax

```
d=vec_permi(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector bool long long	vector bool long long	vector bool long long	0–3
vector signed long long	vector signed long long	vector signed long long	
vector unsigned long long	vector unsigned long long	vector unsigned long long	
vector double	vector double	vector double	

Result value

If we use a[0] and a[1] to represent the first and second eight-byte-long elements in a, and use b[0] and b[1] for elements in b, then this function determines the elements in the result vector based on the binary value of c. This is illustrated as follows:

- 00 - a[0], b[0]
- 01 - a[0], b[1]
- 10 - a[1], b[0]
- 11 - a[1], b[1]

vec_promote
Purpose

Constructs a vector with a in element position b.

Syntax

```
d=vec_promote(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector signed char	signed char	signed int
vector unsigned char	unsigned char	
vector signed short	signed short	
vector unsigned short	unsigned short	
vector signed int	signed int	
vector unsigned int	unsigned int	
vector signed long long	signed long long	
vector unsigned long long	unsigned long	
vector float	float	
vector double	double	

Result value

This function uses the modulo arithmetic on b to determine the element number. For example, if b is out of range, the compiler uses b modulo the number of elements in the vector to determine the element position. The other elements of the vector are undefined.

vec_re

Purpose

Returns a vector containing estimates of the reciprocals of the corresponding elements of the given vector.

Syntax

```
d=vec_re(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector float	vector float
vector double	vector double

Result value

Each element of the result contains the estimated value of the reciprocal of the corresponding element of a.

vec_round

Purpose

Returns a vector containing the rounded values of the corresponding elements of the given vector.

Syntax

```
d=vec_round(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector float	vector float
vector double	vector double

Result value

Each element of the result contains the value of the corresponding element of a, rounded to the nearest representable floating-point integer, using IEEE round-to-nearest rounding.

Note: This function might not follow the strict operation definition of the resolution of a tie during a round when you specify the **-qstrict=nooperationprecision** compiler option.

vec_roundc

Purpose

Returns a vector by rounding every single-precision or double-precision floating-point element in the given vector to integer.

This function uses the current rounding mode. It requires the POWER7 architecture.

Syntax

```
d=vec_roundc(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector float	vector float
vector double	vector double

vec_roundm
Purpose

Returns a vector containing the largest representable floating-point integer values less than or equal to the values of the corresponding elements of the given vector.

Note: `vec_roundm` is another name for `vec_floor`. For details, see "vec_floor" on page 473.

Syntax

```
d=vec_roundm(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector float	vector float
vector double	vector double

vec_roundp
Purpose

Returns a vector containing the smallest representable floating-point integer values greater than or equal to the values of the corresponding elements of the given vector.

Note: `vec_roundp` is another name for `vec_ceil`. For details, see "vec_ceil" on page 464.

Syntax

```
d=vec_roundp(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector float	vector float
vector double	vector double

vec_roundz
Purpose

Returns a vector containing the truncated values of the corresponding elements of the given vector.

Note: `vec_roundz` is another name for `vec_trunc`. For details, see "vec_trunc" on page 494.

Syntax

```
d=vec_roundz(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector float	vector float
vector double	vector double

Result value

Each element of the result contains the value of the corresponding element of a, truncated to an integral value.

vec_rsqrte
Purpose

Returns a vector containing estimates of the reciprocal square roots of the corresponding elements of the given vector.

Syntax

```
d=vec_rsqrte(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector float	vector float
vector double	vector double

Result value

Each element of the result contains the estimated value of the reciprocal square root of the corresponding element of a.

vec_sel
Purpose

Selectively merges two vectors.

Syntax

```
d=vec_sel(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector bool char	vector bool char	vector bool char	vector bool char
			vector unsigned char
vector signed char	vector signed char	vector signed char	vector bool char
			vector unsigned char
vector unsigned char	vector unsigned char	vector unsigned char	vector bool char
			vector unsigned char
vector bool short	vector bool short	vector bool short	vector bool short
			vector unsigned short
vector signed short	vector signed short	vector signed short	vector bool shot
			vector unsigned short
vector unsigned short	vector unsigned short	vector unsigned short	vector bool short
			vector unsigned short
vector bool int	vector bool int	vector bool int	vector bool int
			vector unsigned int
vector signed int	vector signed int	vector signed int	vector bool int
			vector unsigned int
vector unsigned int	vector unsigned int	vector unsigned int	vector bool int
			vector unsigned int
vector bool long long	vector bool long long	vector bool long long	vector bool long long
			vector unsigned long long
vector signed long long	vector signed long long	vector signed long long	vector bool long long
			vector unsigned long long
vector unsigned long long	vector unsigned long long	vector unsigned long long	vector bool long long
			vector unsigned long long
vector float	vector float	vector float	vector bool int
			vector unsigned int
vector double	vector double	vector double	vector bool long long
			vector unsigned long long

Result value

Each bit of the result vector has the value of the corresponding bit of a if the corresponding bit of c is 0, or the value of the corresponding bit of b otherwise.

vec_sl
Purpose

Performs a left shift for each element of a vector.

Syntax

```
d=vec_sl(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector signed char	vector signed char	vector unsigned char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector unsigned short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector unsigned int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector unsigned long long
vector unsigned long long	vector unsigned long long	vector unsigned long long

Result value

Each element of the result vector is the result of left shifting the corresponding element of a by the number of bits specified by the value of the corresponding element of b, modulo the number of bits in the element. The bits that are shifted out are replaced by zeroes.

vec_sldw

Purpose

Shift Left Double by Word Immediate

Returns a vector by concatenating a and b and then left-shifting the result vector by multiples of 4 bytes. c specifies the offset for the shifting operation.

Syntax

```
d=vec_sldw(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b	c
vector bool char	vector bool char	vector bool char	0–3
vector signed char	vector signed char	vector signed char	
vector unsigned char	vector unsigned char	vector unsigned char	
vector bool short	vector bool short	vector bool short	
vector signed short	vector signed short	vector signed short	
vector unsigned short	vector unsigned short	vector unsigned short	
vector bool int	vector bool int	vector bool int	
vector signed int	vector signed int	vector signed int	
vector unsigned int	vector unsigned int	vector unsigned int	
vector bool long long	vector bool long long	vector bool long long	
vector signed long long	vector signed long long	vector signed long long	
vector unsigned long long	vector unsigned long long	vector unsigned long long	
vector float	vector float	vector float	
vector double	vector double	vector double	

Result value

After the left-shifting operation, the two concatenated operands that are in the 32-byte sequence by multiples of 4-bytes specified by operand c, the function takes the four leftmost words and forms a 16-byte result vector.

vec_splat
Purpose

Returns a vector that has all of its elements set to a given value.

Syntax

```
d=vec_splat(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector bool char	vector bool char	0 - 15
vector signed char	vector signed char	0 - 15
vector unsigned char	vector unsigned char	0 - 15
vector bool short	vector bool short	0 - 7
vector signed short	vector signed short	0 - 7
vector unsigned short	vector unsigned short	0 - 7
vector bool int	vector bool int	0–3
vector signed int	vector signed int	0–3

d	a	b
vector unsigned int	vector unsigned int	0–3
vector bool long long	vector bool long long	0–1
vector signed long long	vector signed long long	0–1
vector unsigned long long	vector unsigned long long	0–1
vector float	vector float	0–3
vector double	vector double	0–1

Result value

Assume that the elements of a are numbered beginning with 0. The value of the element of a specified by b is given to each element of the result vector.

vec_splats
Purpose

Constructs a vector and sets the value of all its elements to a.

Syntax

```
d=vec_splats(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector signed char	signed char
vector unsigned char	unsigned char
vector signed short	signed short
vector unsigned short	unsigned short
vector signed int	signed int
vector unsigned int	unsigned int
vector signed long long	signed long long
vector unsigned long long	unsigned long long
vector float	float
vector double	double

vec_sqrt
Purpose

Returns a vector containing the square root of each element in the given vector.

This function requires the POWER7 architecture.

Syntax

```
d=vec_sqrt(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector float	vector float
vector double	vector double

vec_sr

Purpose

Performs a right shift for each element of a vector.

Syntax

```
d=vec_sr(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector signed char	vector signed char	vector unsigned char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector unsigned short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector unsigned int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector unsigned long long
vector unsigned long long	vector unsigned long long	vector unsigned long long

Result value

Each element of the result vector is the result of right shifting the corresponding element of a by the number of bits specified by the value of the corresponding element of b, modulo the number of bits in the element. The bits that are shifted out are replaced by zeroes.

vec_sra

Purpose

Performs an algebraic right shift for each element of a vector.

Syntax

```
d=vec_sra(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector signed char	vector signed char	vector unsigned char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector unsigned short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector unsigned int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector unsigned long long
vector unsigned long long	vector unsigned long long	vector unsigned long long

Result value

Each element of the result vector is the result of algebraically right shifting the corresponding element of a by the number of bits specified by the value of the corresponding element of b, modulo the number of bits in the element. The bits that are shifted out are replaced by copies of the most significant bit of the element of a.

vec_sub
Purpose

Returns a vector containing the differences of each set of corresponding elements of the given vectors.

This function emulates the operation on long long vectors.

Syntax

```
d=vec_sub(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector signed char	vector signed char	vector signed char
vector unsigned char	vector unsigned char	vector unsigned char
vector signed short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short
vector signed int	vector signed int	vector signed int
vector unsigned int	vector unsigned int	vector unsigned int
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector float	vector float

d	a	b
vector double	vector double	vector double

Result value

The value of each element of the result is the result of subtracting the value of the corresponding element of b from the value of the corresponding element of a. The arithmetic is modular for integer vectors.

vec_trunc
Purpose

Returns a vector containing the truncated values of the corresponding elements of the given vector.

Note: `vec_trunc` is another name for `vec_roundz`. For details, see "vec_roundz" on page 486.

vec_xld2
Purpose

Loads a 16-byte vector from two 8-byte elements at the memory address specified by the displacement a and the pointer b.

This function requires the POWER7 architecture.

Syntax

```
d=vec_xld2(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Note: The type for operand a in the following table is: `int` in 32-bit mode, and `long` in 64-bit mode.

d	a	b
vector signed char	int	signed char *
	long	
vector unsigned char	int	unsigned char *
	long	
vector signed short	int	signed short *
	long	
vector unsigned short	int	unsigned short *
	long	
vector signed int	int	signed int *
	long	

d	a	b
vector unsigned int	int	unsigned int *
	long	
vector signed long long	int	signed long long *
	long	
vector unsigned long long	int	unsigned long long *
	long	
vector float	int	float *
	long	
vector double	int	double *
	long	

Result value

This function adds the displacement and the pointer R-value to obtain the address for the load operation. It does not truncate the affected address to a multiple of 16 bytes.

vec_xlds

Purpose

Loads an 8-byte element from the memory address specified by the displacement a and the pointer b and then splats it onto a vector.

This function requires the POWER7 architecture.

Syntax

```
d=vec_xlds(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Note: The type for operand a in the following table is: `int` in 32-bit mode, and `long` in 64-bit mode.

d	a	b
vector signed long long	int	signed long long *
	long	
vector unsigned long long	int	unsigned long long *
	long	
vector double	int	double *
	long	

Result value

This function adds the displacement and the pointer R-value to obtain the address for the load operation. It does not truncate the affected address to a multiple of 16 bytes.

vec_xlw4
Purpose

Loads a 16-byte vector from four 4-byte elements at the memory address specified by the displacement a and the pointer b.

This function requires the POWER7 architecture.

Syntax

```
d=vec_xlw4(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Note: The type for operand a in the following table is: `int` in 32-bit mode, and `long` in 64-bit mode.

d	a	b
vector signed char	int	signed char *
	long	
vector unsigned char	int	unsigned char *
	long	
vector signed short	int	signed short *
	long	
vector unsigned short	int	unsigned short *
	long	
vector signed int	int	signed int *
	long	
vector unsigned int	int	unsigned int *
	long	
vector float	int	float *
	long	

Result value

This function adds the displacement and the pointer R-value to obtain the address for the load operation. It does not truncate the affected address to a multiple of 16 bytes.

vec_xor

Purpose

Performs a bitwise XOR of the given vectors.

Syntax

d=vec_xor(a, b)

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector bool char	vector bool char	vector bool char
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector bool short	vector bool short	vector vector bool short
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector bool int	vector bool int	vector bool int
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector bool long long	vector signed long long
	vector signed long long	vector signed long long
		vector bool long long
vector unsigned long long	vector bool long long	vector unsigned long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
vector float	vector bool int	vector float
	vector float	vector bool int
		vector float

d	a	b
vector double	vector bool long long	vector double
	vector double	vector bool long long
		vector double

Result value

The result is the bitwise XOR of a and b.

vec_xstd2

Purpose

Puts a 16-byte vector a as two 8-byte elements to the memory address specified by the displacement b and the pointer c.

This function requires the POWER7 architecture.

Syntax

```
d=vec_xstd2(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Note: The type for operand a in the following table is: int in 32-bit mode, and long in 64-bit mode.

d	a	b	c
void	vector signed char	int	signed char *
		long	
	vector unsigned char	int	unsigned char *
		long	
	vector signed short	int	signed short *
		long	
	vector unsigned short	int	unsigned short *
		long	
	vector signed int	int	signed int *
		long	
	vector unsigned int	int	unsigned int *
		long	
	vector signed long long	int	signed long long *
		long	
	vector unsigned long long	int	unsigned long long *
		long	
	vector float	int	float *
		long	
	vector double	int	double *
		long	
	vector pixel	int	signed short *
			unsigned short *
		long	signed short *
			unsigned short *

Result value

This function adds the displacement and the pointer R-value to obtain the address for the store operation. It does not truncate the affected address to a multiple of 16 bytes.

vec_xstw4
Purpose

Puts a 16-byte vector a to four 4-byte elements at the memory address specified by the displacement b and the pointer c.

This function requires the POWER7 architecture.

Syntax

```
d=vec_xstw4(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Note: The type for operand b in the following table is: `int` in 32-bit mode, and `long` in 64-bit mode.

d	a	b	c
void	vector signed char	int	signed char *
		long	
	vector unsigned char	int	unsigned char *
		long	
	vector signed short	int	signed short *
		long	
	vector unsigned short	int	unsigned short *
		long	
	vector signed int	int	signed int *
		long	
	vector unsigned int	int	unsigned int *
		long	
	vector float	int	float *
		long	
	vector pixel	int	signed short *
			unsigned short *
		long	signed short *
			unsigned short *

Result value

This function adds the displacement and the pointer R-value to obtain the address for the store operation. It does not truncate the affected address to a multiple of 16 bytes.

Miscellaneous built-in functions

Miscellaneous functions are grouped into the following categories:

- "Optimization-related functions" on page 501
- "Move to/from register functions" on page 502
- "Memory-related functions" on page 504

Optimization-related functions

__alignx
Purpose

Allows for optimizations such as automatic vectorization by informing the compiler that the data pointed to by *pointer* is aligned at a known compile-time offset.

Prototype

> void __alignx (int *alignment*, const void* *pointer*);

Parameters

alignment
    Must be a constant integer with a value greater than zero and of a power of two.

__builtin_expect
Purpose

Indicates that an expression is likely to evaluate to a specified value. The compiler may use this knowledge to direct optimizations.

Prototype

> long __builtin_expect (long *expression*, long *value*);

Parameters

expression
    Should be an integral-type expression.

value
    Must be a constant literal.

Usage

If the *expression* does not actually evaluate at run time to the predicted value, performance may suffer. Therefore, this built-in function should be used with caution.

__fence
Purpose

Acts as a barrier to compiler optimizations that involve code motion, or reordering of machine instructions. Compiler optimizations will not move machine instructions past the location of the __fence call.

Prototype

> void __fence (void);

Examples

This function is useful to guarantee the ordering of instructions in the object code generated by the compiler when optimization is enabled.

Move to/from register functions

__mftb
Purpose

Move from Time Base

In 32-bit compilation mode, returns the lower word of the time base register. In 64-bit mode, returns the entire doubleword of the time base register.

Prototype

unsigned long __mftb (void);

Usage

In 32-bit mode, this function can be used in conjunction with the __mftbu built-in function to read the entire time base register. In 64-bit mode, the entire doubleword of the time base register is returned, so separate use of __mftbu is unnecessary

It is recommended that you insert the __fence built-in function before and after the __mftb built-in function.

__mftbu
Purpose

Move from Time Base Upper

Returns the upper word of the time base register.

Prototype

unsigned int __mftbu (void);

Usage

In 32-bit mode you can use this function in conjunction with the __mftb built-in function to read the entire time base register

It is recommended that you insert the __fence built-in function before and after the __mftbu built-in function.

__mfmsr
Purpose

Move from Machine State Register

Moves the contents of the machine state register (MSR) into bits 32 to 63 of the designated general-purpose register.

Prototype

> unsigned long __mfmsr (void);

Usage

Execution of this instruction is privileged and restricted to supervisor mode only.

__mfspr
Purpose

Move from Special-Purpose Register

Returns the value of given special purpose register.

Prototype

> unsigned long __mfspr (const int *registerNumber*);

Parameters

registerNumber
> The number of the special purpose register whose value is to be returned. The *registerNumber* must be known at compile time.

__mtmsr
Purpose

Move to Machine State Register

Moves the contents of bits 32 to 63 of the designated GPR into the MSR.

Prototype

> void __mtmsr (unsigned long);

Usage

Execution of this instruction is privileged and restricted to supervisor mode only.

__mtspr
Purpose

Move to Special-Purpose Register

Sets the value of a special purpose register.

Prototype

> void __mtspr (const int *registerNumber*, unsigned long *value*);

Parameters

registerNumber
> The number of the special purpose register whose value is to be set. The *registerNumber* must be known at compile time.

value
>   Must be known at compile time.

Memory-related functions

__alloca
Purpose

Allocates space for an object. The allocated space is put on the stack and freed when the calling function returns.

Prototype

>   void* __alloca (size_t *size*)

Parameters

size
>   An integer representing the amount of space to be allocated, measured in bytes.

__builtin_frame_address, __builtin_return_address
Purpose

Returns the address of the stack frame, or return address, of the current function, or of one of its callers.

Prototype

>   void* __builtin_frame_address (unsigned int *level*);

>   void* __builtin_return_address (unsigned int *level*);

Parameters

level
>   A constant literal indicating the number of frames to scan up the call stack. The *level* must range from 0 to 63. A value of 0 returns the frame or return address of the current function, a value of 1 returns the frame or return address of the caller of the current function and so on.

Return value

Returns 0 when the top of the stack is reached. Optimizations such as inlining may affect the expected return value by introducing extra stack frames or fewer stack frames than expected. If a function is inlined, the frame or return address corresponds to that of the function that is returned to.

__mem_delay
Purpose

The **__mem_delay** built-in function specifies how many delay cycles there are for specific loads. These specific loads are delinquent loads with a long memory access latency because of cache misses.

When you specify which load is delinquent the compiler takes that information and carries out optimizations such as data prefetching. In addition, when you run

-qprefetch=assistthread, the compiler uses the delinquent load information to perform analysis and generate prefetching assist threads. For more information, see "-qprefetch" on page 245.

Prototype

> void* __mem_delay (const void *address, const unsigned int *cycles*);

Parameters

address
> The address of the data to be loaded or stored.

cycles
> A compile time constant, typically either L1 miss latency or L2 miss latency.

Usage

The **__mem_delay** built-in function is placed immediately before a statement that contains a specified memory reference.

Examples

Here is how you generate code using assist threads with **__mem_delay**:

Initial code:
```
int y[64], x[1089], w[1024];

  void foo(void){
    int i, j;
    for (i = 0; i &l; 64; i++) {
      for (j = 0; j < 1024; j++) {

        /* what to prefetch? y[i]; inserted by the user */
        __mem_delay(&y[i], 10);
        y[i] = y[i] + x[i + j] * w[j];
        x[i + j + 1] = y[i] * 2;
    }
  }
}
```

Assist thread generated code:
```
void foo@clone(unsigned thread_id, unsigned version)

{ if (!1) goto lab_1;

/* version control to synchronize assist and main thread */
if (version == @2version0) goto lab_5;

goto lab_1;

lab_5:

@CIV1 = 0;

do { /* id=1 guarded */ /* ~2 */

if (!1) goto lab_3;

@CIV0 = 0;

do { /* id=2 guarded */ /* ~4 */
```

```
/* region = 0 */

/* __dcbt call generated to prefetch y[i] access */
__dcbt(((char *)&y + (4)*(@CIV1)))
@CIV0 = @CIV0 + 1;
} while ((unsigned) @CIV0 < 1024u); /* ~4 */

lab_3:
@CIV1 = @CIV1 + 1;
} while ((unsigned) @CIV1 < 64u); /* ~2 */

lab_1:

return;
}
```

Related information
- "-qprefetch" on page 245

Built-in functions for parallel processing

Use these built-in functions to obtain information about the parallel environment:
- "OpenMP built-in functions"

OpenMP built-in functions

Function definitions for the omp_ functions can be found in the omp.h header file.

For complete information about OpenMP runtime library functions, refer to the OpenMP C/C++ Application Program Interface specification in www.openmp.org.

Related information
- "Environment variables for parallel processing" on page 25

omp_get_max_active_levels
Purpose

Retrieves the value of the *max-active-levels-var* internal control variable that determines the maximum number of nested active parallel regions. *max-active-levels-var* can be set with the *OMP_MAX_ACTIVE_LEVELS* environment variable or the omp_set_max_active_levels function.

Prototype

int omp_get_max_active_levels(void);

omp_set_max_active_levels
Purpose

Sets the value of the *max-active-levels-var* internal control variable to the value in the argument. If the number of parallel levels requested exceeds the number of the supported level of parallelism, the value of *max-active-levels-var* is set to the number of parallel levels supported by the runtime. If the number of parallel levels requested is not a positive integer, this routine call is ignored. When the nested parallelism is turned off, this routine has no effects and the value of *max-active-levels-var* remains 1. *max-active-levels-var* can also be set with the *OMP_MAX_ACTIVE_LEVELS* environment variable. To retrieve the value for *max-active-levels-var*, use the *omp_get_max_active_levels* function.

Prototype

    void omp_set_max_active_levels(int max_levels);

omp_get_schedule
Purpose

Returns the *run-sched-var* internal control variable of the team that is processing the parallel region. The argument *kind* returns the type of schedule that will be used. *modifier* represents the chunk size that is set for applicable schedule types. *run-sched-var* can be set with the *OMP_SCHEDULE* environment variable or the **omp_set_schedule** function.

Prototype

    int omp_get_schedule(omp_sched_t * kind, int * modifier);

Parameters

kind
    The value returned for *kind* is one of the schedule types affinity, auto, dynamic, guided, runtime, or static.

modifier
    For the schedule type dynamic, guided, or static, modifier is the chunk size that is set. For the schedule type auto, modifier has no meaning.

Related reference

"omp_set_schedule"

Related information

"OMP_SCHEDULE=*algorithm* environment variable" on page 29

omp_set_schedule
Purpose

Sets the value of the *run-sched-var* internal control variable. Use **omp_set_schedule** if you want to set the schedule type separately from the *OMP_SCHEDULE* environment variable.

Prototype

    void omp_set_schedule (omp_sched_t kind, int modifier);

Parameters

kind
    Must be one of the schedule types affinity, auto, dynamic, guided, runtime, or static.

modifier
    For the schedule type dynamic, guided, orstatic, modifier is the chunk size that you want to set. Generally it is a positive integer. If the value is less than one, the default will be used. For the schedule type auto, modifier has no meaning.

Related information

"OMP_SCHEDULE=*algorithm* environment variable" on page 29

omp_get_thread_limit
Purpose

Retrieves the maximum number of OpenMP threads stored in the *thread-limit-var* internal control variable that are available to the program. *thread-limit-var* can be set with the *OMP_THREAD_LIMIT* environment variable.

Prototype

```
int omp_get_thread_limit(void);
```

omp_get_level
Purpose

Use **omp_get_level** to return the number of active and inactive nested parallel regions that the generating task is executing in. This does not include the implicit parallel region.

Prototype

```
int omp_get_level(void);
```

omp_get_ancestor_thread_num
Purpose

Use **omp_get_ancestor_thread_num** to return the thread number in the current level of the ancestor that is at the specified nested level. **omp_get_ancestor_thread_num** returns -1 if the nested level is not within the range of 0 and the current thread's nested level as returned by **omp_get_level**.

Prototype

```
int omp_get_ancestor_thread_num(int level);
```

omp_get_team_size
Purpose

Use **omp_get_team_size** to return the thread team size that the ancestor belongs to. **omp_get_team_size** returns -1 if the nested level is not within the range of 0 and the current thread's nested level as returned by **omp_get_level**.

Prototype

```
int omp_get_team_size(int level);
```

omp_get_active_level
Purpose

Use **omp_get_active_level** to return the number of active parallel regions that are nested.

Prototype

> int omp_get_active_level(void);

omp_get_num_threads
Purpose

Returns the number of threads currently in the team executing the parallel region from which it is called.

Prototype

> int omp_get_num_threads (void);

omp_set_num_threads
Purpose

Overrides the setting of the OMP_NUM_THREADS environment variable, and specifies the number of threads to use in parallel regions following this directive.

Prototype

> void omp_set_num_threads (int *num_threads*);

Parameters

num_threads
> Must be a positive integer.

Usage

If the num_threads clause is present, then for the parallel region it is applied to, it supersedes the number of threads requested by this function or the OMP_NUM_THREADS environment variable. Subsequent parallel regions are not affected by it.

omp_get_max_threads
Purpose

Returns the maximum value that can be returned by calls to omp_get_num_threads.

Prototype

> int omp_get_max_threads (void);

omp_get_thread_num
Purpose

Returns the thread number, within its team, of the thread executing the function.

Prototype

> int omp_get_thread_num (void);

Return value

The thread number lies between 0 and `omp_get_num_threads()`-1, inclusive. The master thread of the team is thread 0.

omp_get_num_procs
Purpose

Returns the maximum number of processors that could be assigned to the program.

Prototype

    int omp_get_num_procs (void);

omp_in_parallel
Purpose

Returns non-zero if it is called within the dynamic extent of a parallel region executing in parallel; otherwise, returns 0.

Prototype

    int omp_in_parallel (void);

omp_set_dynamic
Purpose

Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.

Prototype

    void omp_set_dynamic (int dynamic_threads);

omp_get_dynamic
Purpose

Returns non-zero if dynamic thread adjustments enabled and returns 0 otherwise.

Prototype

    int omp_get_dynamic (void);

omp_set_nested
Purpose

Enables or disables nested parallelism.

Prototype

    void omp_set_nested (int);

Return value

In the current implementation, nested parallel regions are always serialized. As a result, has no effect.

omp_get_nested
Purpose

Returns non-zero if nested parallelism is enabled and 0 if it is disabled.

Prototype

int omp_get_nested (void);

Return value

In the current implementation, nested parallel regions are always serialized. As a result, always returns 0.

omp_init_lock, omp_init_nest_lock
Purpose

Initializes the lock associated with the parameter *lock* for use in subsequent calls.

Prototype

void omp_init_lock (omp_lock_t *_lock_*);

void omp_init_nest_lock (omp_nest_lock_t *_lock_*);

omp_destroy_lock, omp_destroy_nest_lock
Purpose

Ensures that the specified lock variable *lock* is uninitialized.

Prototype

void omp_destroy_lock (omp_lock_t *_lock_*);

void omp_destroy_nest_lock (omp_nest_lock_t *_lock_*);

omp_set_lock, omp_set_nest_lock
Purpose

Blocks the thread executing the function until the specified lock is available and then sets the lock.

Prototype

void omp_set_lock (omp_lock_t * *lock*);

void omp_set_nest_lock (omp_nest_lock_t * *lock*);

Usage

A simple lock is available if it is unlocked. A nestable lock is available if it is unlocked or if it is already owned by the thread executing the function.

omp_unset_lock, omp_unset_nest_lock
Purpose

Releases ownership of a lock.

Prototype

void omp_unset_lock (omp_lock_t * *lock*);

void omp_unset_nest_lock (omp_nest_lock_t * *lock*);

omp_test_lock, omp_test_nest_lock
Purpose

Attempts to set a lock but does not block execution of the thread.

Prototype

int omp_test_lock (omp_lock_t * *lock*);

int omp_test_nest_lock (omp_nest_lock_t * *lock*);

omp_get_wtime
Purpose

Returns the time elapsed from a fixed starting time.

Prototype

double omp_get_wtime (void);

Usage

The value of the fixed starting time is determined at the start of the current program, and remains constant throughout program execution.

omp_get_wtick
Purpose

Returns the number of seconds between clock ticks.

Prototype

double omp_get_wtick (void);

Usage

The value of the fixed starting time is determined at the start of the current program, and remains constant throughout program execution.

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.

513

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
8200 Warden Avenue
Markham, Ontario   L6G 1C7
Canada

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

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

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

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

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. 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, 2010. All rights reserved.

Trademarks and service marks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

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

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

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

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

Index

Special characters

IBM ®

Program Number: 5724-X14

Printed in USA