IBM XL C/C++ for AIX, V10.1

Compiler Reference

Version 10.1

IBM XL C/C++ for AIX, V10.1

Compiler Reference

Version 10.1

First edition

This edition applies to IBM XL C/C++ for AIX, V10.1 (Program number 5724-U81) 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

About this information

This information is a reference for the IBM® XL C/C++ for AIX®, V10.1 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 3, "Compiler options reference," on page 39 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 4, "Compiler pragmas reference," on page 309 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 and SMP directives.

Chapter 5, "Compiler predefined macros," on page 379 provides a list of compiler macros according to category.

Chapter 6, "Compiler built-in functions," on page 391 contains individual descriptions of XL C/C++built-in functions for PowerPC® architectures, categorized by their functionality.

Conventions

Typographical conventions

The following table explains the typographical conventions used in the IBM XL C/C++ for AIX, V10.1 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.

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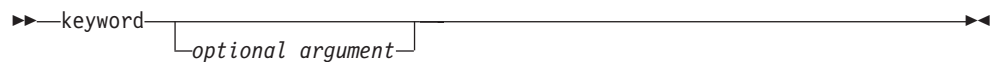
- If you can choose from two or more items, they are shown vertically, in a stack.

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

```
►►──keyword──┬─required_argument1─┬──────────────────────────►◄
             └─required_argument2─┘
```
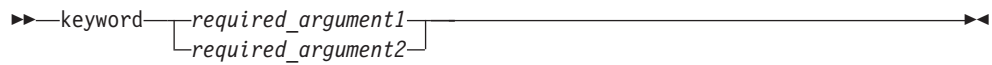
If choosing one of the items is optional, the entire stack is shown below the main path.

```
►►──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─┴──────────────────────────►◄
```
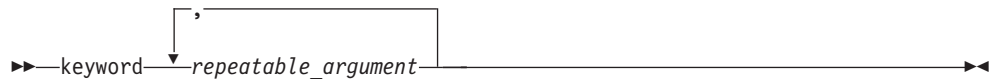
- The item that is the default is shown above the main path.

```
          ┌─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─"─┘
```
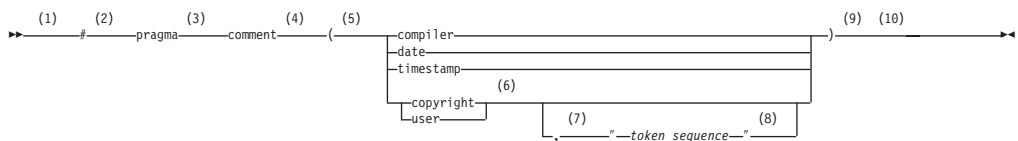
Notes:

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 AIX, V10.1 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 AIX, V10.1 Installation Guide*.

  The information center is viewable on the Web at http://publib.boulder.ibm.com/infocenter/comphelp/v101v121/index.jsp.

- PDF documents

  PDF documents are located by default in the /usr/vacpp/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/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 AIX, V10.1 Installation Guide, GC23-8889-00	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 AIX, V10.1, GC23-8897-00	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 AIX, V10.1 Compiler Reference, SC23-8886-00	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 AIX, V10.1 Language Reference, SC23-8888-00	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 AIX, V10.1 Optimization and Programming Guide, SC23-8887-00	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.
Standard C++ Library Reference, SC23-8890-00	standlib.pdf	Contains reference information about the standard C++ runtime libraries and headers.
C/C++ Legacy Class Libraries Reference, SC09-7652-00	legacy.pdf	Contains reference information about the USL I/O Stream Library and the Complex Mathematics Library.

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 redbooks, white papers, tutorials, and other articles, is available on the Web at:

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

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.
- *Information Technology – Programming Languages – Extension for the programming language C to support decimal floating-point arithmetic, ISO/IEC WDTR 24732*. This draft technical report has been submitted to the C standards committee, and is available at http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1176.pdf.
- *Decimal Types for C++: Draft 4* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1977.html

Other IBM information

- *AIX Commands Reference, Volumes 1 - 6, SC23-4888*
- *Technical Reference: Base Operating System and Extensions, Volumes 1 & 2, SC23-4913*
- *AIX National Language Support Guide and Reference, SC23-4902*
- *AIX General Programming Concepts: Writing and Debugging Programs, SC23-4896*
- *AIX Assembler Language Reference, SC23-4923*

  All AIX information is available at http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp.
- *Parallel Environment for AIX: Operation and Use*
- *ESSL for AIX V4.2 Guide and Reference, SA22-7904*, available at http://publib.boulder.ibm.com/clresctr/windows/public/esslbooks.html

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/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.

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 invoke 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 11
- "Preprocessing" on page 12
- "Linking" on page 14
- "Compiler messages and listings" on page 17

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 that for each invocation command, the compiler configuration file defines default option settings and, in some cases, macros; for information on the defaults implied by a particular invocation, see the /etc/vac.cfg file for your system.

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, xlc128_r4, xlc128_r7, xlc128, xlc128_r, xlc128_r4, xlc128_r7
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, c99_r4, c99_r7, c99_128, c99_128_r, c99_128_r4, c99_128_r7

1

Table 4. Compiler invocations (continued)

Basic invocations	Description	Equivalent special invocations
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, c89_r4, c89_r7, c89_128, c89_128_r, c89_128_r4, c89_128_r7
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, cc_r4, cc_r7, cc128, cc128_r, cc128_r4, cc128_r7
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 11.	
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++_r4, xlc++_r7, xlc++128, xlc++128_r, xlc++128_r4, xlc++128_r7, xlC_r, xlC_r4, xlC_r7, xlC128, xlC128_r, xlC128_r4, xlC128_r7
xlc++core, xlCcore	Invokes the compiler as described above for **xlc++** and **xlC**, but links only to the core of the runtime library. Use this invocation if you want to link your application to a runtime library other than that supplied with XL C++.	xlc++core_r, xlc++core_r7, xlc++core128, xlc++core128_r, xlc++core128_r7, xlCcore_r, xlCcore_r7, xlC128core, xlC128core_r, xlC128core_r7
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 11.	

Table 5. Suffixes for special invocations

128-suffixed invocations	All **128**-suffixed invocation commands are functionally similar to their corresponding base compiler invocations. They specify the **-qldbl128** option, which increases the length of long double types in your program from 64 to 128 bits. They also link with the 128-bit versions of the C and C++ runtime libraries.
_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. The **_r7** invocations are provided to help migrate programs based on Posix Draft 7 to Posix Draft 10. The **_r4** invocations should be used for DCE 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 ─┘
```
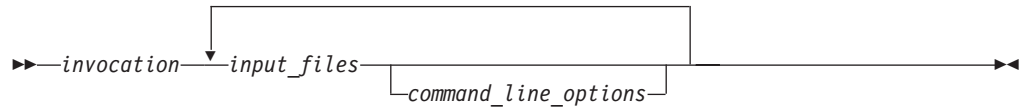
The parameters of the compiler invocation command can be the names of input
files, compiler options, and linker options.

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 appear. 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 will not be run
and temporary object files will be removed.

By default, the compiler preprocesses and compiles all the specified source files.
Although you will 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

Extended Common Object File Format (XCOFF) files that have not been stripped with the operating system **strip** command can be used as input to the compiler. See the **strip** command in the *AIX Commands Reference* and the description of a.out file format in the *AIX Files Reference* for more information.

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.

The format of the a.out file is described in the *AIX Files Reference*.

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 shr.o, 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*.u 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. XL C/C++ 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 will override compiler options specified on the command line.

2. Compiler options specified on the command line will 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 option appearing later in the invocation takes precedence.

3. Compiler options specified as environment variables will override compiler options specified in a configuration file.

4. Compiler options specified in a configuration file, command line or source program will 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 ────┤
                                   └──────── : ◄───────┘
```
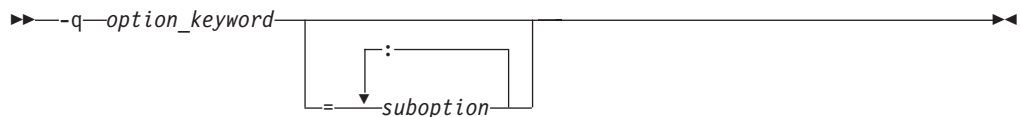
Command-line options in the **-q***option_keyword* format are similar to on and off switches. For *most* **-q** options, if a given option is specified more than once, the last appearance of that option on the command line is the one recognized by the compiler. For example, **-qsource** turns on the 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 (/etc/vac.cfg) 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, there are several ways to specify the syntax of the pragmas:

- Using **#pragma options** *option_name* syntax — Many command-line options allow you to use 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; also, for a complete list of these, see "#pragma options" on page 340.

- 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 55, 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 309.

Other pragmas do not have equivalent command-line options; these are described in detail throughout Chapter 4, "Compiler pragmas reference," on page 309.

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 above rules. The table below 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
-qfloat=hssngl	-qfloat=spnans	-qfloat=hssngl
-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	-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. OBJECT_MODE environment variable setting
3. Configuration file settings
4. Command line compiler options (**-q32**, **-q64**, **-qarch**, **-qtune**)
5. 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. If neither of these compiler options is set, the compiler mode is set by the value of the OBJECT_MODE environment variable. If the OBJECT_MODE environment variable is also not set, the compiler assumes 32-bit compilation mode.
- *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 67 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 283.

Possible option conflicts and compiler resolution of these conflicts are described below:

- **-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 OBJECT_MODE environment variable or **-q32**/**-q64** compiler settings.

Related information
- "-qarch" on page 67
- "-qtune" on page 283
- "-q32, -q64" on page 58

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 then 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 /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:

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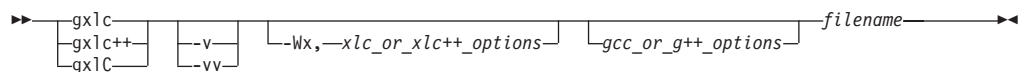
where:

filename
    Is the name of the file to be compiled.

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

-vv   Allows you to run a simulation. 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 to be 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 101	Preprocesses the source files and writes the output to standard output. By default, `#line` directives are generated.
"-P" on page 213	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 222	Toggles on and off the generation of `#line` directives for the **-E** and **-P** options.
"-C, -C!" on page 80	Preserves comments in preprocessed output.
"-D" on page 92	Defines a macro name from the command line, as if in a `#define` directive.
"-U" on page 287	Undefines a macro name defined by the compiler or by the **-D** option.
"-qshowmacros" on page 245	Emits macro definitions to preprocessed output.

Directory search sequence for include files

XL C/C++ 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 appears 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 /usr/vacpp/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 /usr/vacpp//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. This is normally /usr/vacpp/include/, but the search path can be changed with the **-qc_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. This is normally /usr/vacpp//include/ but the search path can be changed with the **-qcpp_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 135
- "-qc_stdinc (C only)" on page 90
- "-qcpp_stdinc (C++ only)" on page 91
- "-qidirfirst" on page 136
- "-qinclude" on page 139
- "-qstdinc" on page 259

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 shr.o. 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 52.

Related information

"-qmkshrobj" on page 196

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 52
- "Redistributable libraries"
- **ld** in the *AIX Commands Reference, Volume 5: s through u*

Redistributable libraries

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

- You can ship the filesets that contain the redistributable libraries with the application. The filesets are stored under the runtime/ directory on the installation CD.

- The user can download the filesets that contain the redistributable libraries from the XL C/C++ support Web site at:

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

For information on the licensing requirements related to the distribution of these filesets refer to LicAgree.pdf on the CD.

Table 6. Redistributable libraries

Fileset	Libraries (and default installation path)	Description
xlC.rte	/usr/lpp/xlC/lib/ libibmcls.a/usr/lpp/xlC/lib/ libibmuis.a	XL C++ runtime environment
xlC.aix50	/usr/lpp/xlC/lib/aix53/ libC.a /usr/lpp/xlC/lib/ aix53/libC128.a /usr/lpp/xlC/lib/profiled/ aix53/libC.a /usr/lpp/xlC/lib/profiled/ aix53/libC128.a	XL C++ runtime environment for AIX 5.2
xlC.msg.en_US.rte	/usr/lib/nls/msg/en_US/ ibmcl.cat	XL C++ runtime messages (English)
xlC.msg.ja_JP.rte	/usr/lib/nls/msg/ja_JP/ ibmcl.cat	XL C++ runtime messages (Japanese, IBM-eucJP)
xlC.msg.Ja_JP.rte	/usr/lib/nls/msg/Ja_JP/ ibmcl.cat	XL C++ runtime messages (Japanese, IBM-943)
xlsmp.rte	/usr/include/omp.h/usr/ lpp/xlsmp/default_msg/ smprt.cat	SMP runtime environment
xlsmp.aix52.rte	/usr/lpp/xlsmp/aix53/ libxlomp_ser.a /usr/lpp/xlsmp/aix53/ libxlsmp.a /usr/lpp/xlsmp/aix53/ libxlsmpdebug.a	SMP runtime environment for AIX
xlsmp.msg.en_US.rte	/usr/lib/nls/msg/en_US/ smprt.cat	SMP runtime messages (English, ISO8859-1)
xlsmp.msg.EN_US.rte	/usr/lib/nls/msg/EN_US/ smprt.cat	SMP runtime messages (English, UTF-8)

Table 6. Redistributable libraries (continued)

Fileset	Libraries (and default installation path)	Description
xlsmp.msg.ja_JP.rte	/usr/lib/nls/msg/ja_JP/ smprt.cat	SMP runtime messages (Japanese, IBM-eucJP)
xlsmp.msg.Ja_JP.rte	/usr/lib/nls/msg/Ja_JP/ smprt.cat	SMP runtime messages (Japanese, IBM-943)
xlsmp.msg.JA_JP.rte	/usr/lib/nls/msg/JA_JP/ smprt.cat	SMP runtime messages (Japanese, UTF-8)
xlsmp.msg.zh_CN.rte	/usr/lib/nls/msg/zh_CN/ smprt.cat	SMP runtime messages (Chinese, IBM-eucCN)
xlsmp.msg.ZH_CN.rte	/usr/lib/nls/msg/ZH_CN/ smprt.cat	SMP runtime messages (Chinese, UTF-8)
xlsmp.msg.Zh_CN.rte	/usr/lib/nls/msg/Zh_CN/ smprt.cat	SMP runtime messages (Chinese, GBK)
vac.aix52.lib	/usr/vac/lib/aix53/libxl.a /usr/vac/lib/aix53/ libxlopt.a	XL C libraries for AIX
vacpp.cmp.rte	/usr/vacpp/lib/libC.a /usr/vacpp/lib/libC128.a /usr/vacpp/lib/libC128_r.a /usr/vacpp/lib/libC_r.a /usr/vacpp/lib/profiled/ libC.a /usr/vacpp/lib/ profiled/libC128.a /usr/vacpp/lib/profiled/ libC128_r.a /usr/vacpp/lib/profiled/ libC_r.a	XL C++ compiler application runtime libraries
vacpp.cmp.tools	/usr/vacpp/bin/c++filt /usr/vacpp/bin/linkxlC /usr/vacpp/bin/ makeC++SharedLib /usr/vacpp/bin/ makeC++SharedLib_r /usr/vacpp/bin/ makeC++SharedLib128 /usr/vacpp/bin/ makeC++SharedLib_r7 /usr/vacpp/exe/aix53/ munch	XL C++ utilities
vacpp.memdbg.aix52.rte	/usr/vacpp/lib/aix53/ libhC.a /usr/vacpp/lib/ profiled/aix53/libhC.a /usr/vacpp/lib/aix53/ libhC_r.a /usr/vacpp/lib/profiled/ aix53/libhC_r.a	User heap/memory debug toolkit for AIX

Table 6. Redistributable libraries (continued)

Fileset	Libraries (and default installation path)	Description
vacpp.memdbg.rte	/usr/vacpp/lib/libhm.a/ usr/vacpp/lib/libhm_r.a /usr/vappc/lib/libhmd.a/ usr/vacpp/lib/libhmd_r.a /usr/vacpp/lib/libhmu.a /usr/vacpp/lib/libhmu_r.a /usr/vacpp/lib/libhu.a /usr/vacpp/lib/libhu_r.a usr/vacpp/lib/profiled/ libhm.a /usr/vacpp/lib/ profiled/libhm_r.a /usr/vacpp/lib/profiled/ libhmd.a /usr/vacpp/lib/profiled/ libhmd_r.a /usr/vacpp/lib/profiled/ libhmu.a /usr/vacpp/lib/profiled/ libhmu_r.a /usr/vacpp/lib/profiled/ libhu.a /usr/vacpp/lib/ profiled/libhu_r.a	User heap/memory debug toolkit

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 19
- "Compiler listings" on page 20
- "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 and, if you compile with the **-qsource** option, to a listing file. Note that 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 251
- "-qsrcmsg (C only)" on page 255

- "-qflag" on page 113
- "-w" on page 298
- "-qinfo" on page 141

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
47	- message specific to the C++ linkage helper
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

XL C/C++ uses a multi-level 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. Note that you can adjust the default compiler response by using any of the following options:

- **-qhalt** allows you to halt the compilation phase at a lower severity level than the default
- **-qmaxerr** allows you to halt the compilation phase as soon as a specific number of errors at a specific severity level is reached

- **C++** **-qhaltonmsg** allows you to halt 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.
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 129
- "-qmaxerr" on page 190
- "-qhaltonmsg (C++ only)" on page 130
- 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 may also be displayed for runtime errors. For example, a runtime return code of 99 indicates that a static initialization has failed.

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

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, and release, as well as 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.

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 should 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 /usr/lib/nls/msg/$LANG/*.cat
```

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

The compiler calls the default message catalogs in `/usr/vacpp/exe/default_msg/` when:
- The message catalogs for the locale specified by LANG cannot be found.
- The locale has never been changed from the default, **C**.

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 one of the following messages:

```
1501-229 Compilation ended due to lack of space.
1501-224 fatal error in ../exe/xlCcode: signal 9 received.
```

If lack of paging space causes other compiler programs to fail, the following message is displayed:

```
Killed.
```

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

To check the current paging-space settings enter the command: **lsps -a** or use the AIX System Management Interface Tool (SMIT) command **smit pgsp**.

See your operating system documentation for more information about paging space and how to allocate it.

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 may 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, /etc/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*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 so on. 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 /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" on page 25

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 AIX 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.

LIBPATH
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.

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*.

OBJECT_MODE
Optionally specifies the bit mode for compilation to either 32 or 64 bits. This is equivalent to the **-q32** and **-q64** compiler options. Set the OBJECT_MODE environment variable to a value of 32 for 32-bit compilation mode, or 64 for 64-bit compilation mode. If unspecified, the default compilation mode is 32 bits. See also "-q32, -q64" on page 58 for more information.

PATH Specifies the directory search path for the executable files of the compiler. Executables are in /usr/vac/bin/ and /usr/vacpp/bin/ if installed to the default location.

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.

LIBPATH

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.

MALLOCALIGN=16

Specifies that dynamic memory allocations return 16-byte aligned addresses. See also "-qipa" on page 149.

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 215 for more information.

XL_NOCLONEARCH

If you have compiled a program with the **-qipa=clonearch** option, which generates multiple versions of functions optimized for different runtime architectures, you can set this environment variable to instruct the compiled application to only execute the generic code, that is, the code that is not versioned for a specific architecture. This variable is unset by default; you can set it to help debug your application. See "-qipa" on page 149 for more information.

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 threadsafe compiler mode invocations when compiling parallelized program code.

Related information
- "Pragma directives for parallel processing" on page 359
- "Built-in functions for parallel processing" on page 441

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─┬──────┬─►◄
                 └─ " ─┘                                           └─ " ─┘
```
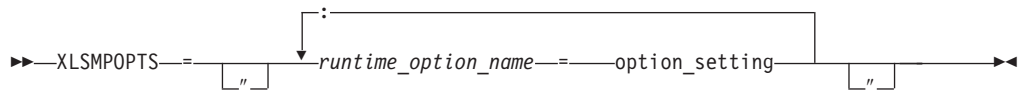
You can specify option names and settings in uppercase or lowercase. You can add blanks before and after the colons and equal signs to improve readability. However, if the XLSMPOPTS option string contains imbedded blanks, you must enclose the entire option string in double quotation marks (").

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 will need. The default value for *num* is 4194304.

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 247 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.

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 on the OpenMP specification, see: www.openmp.org/specs.

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 443
"omp_get_schedule" on page 443

Parallel environment variables:
OMP_NUM_THREADS=*num*

The **OMP_NUM_THREADS** 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_NESTED=TRUE | FALSE

The **OMP_NESTED** 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 may employ additional threads as available. However, no new team will be 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 set to **TRUE**, the number of threads available for executing parallel regions may be adjusted at run time to make the best use of system resources. See the description for **profilefreq=***num* in "XLSMPOPTS" on page 25 for more information.

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

The default setting is **TRUE**.

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:

```
           ┌─PASSIVE─┐
►►──OMP_WAIT_POLICY=──┴─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 will consume 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, then **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 will set 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 will set 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* will be set by the runtime.

►►──OMP_MAX_ACTIVE_LEVELS=n──►◄

Using custom compiler configuration files

XL C/C++ generates a default configuration file /etc/vac.cfg.*nn* at installation time, where *nn* indicates which OS version the configuration file is for. 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 may want to leave the default configuration file as it is.

Otherwise, especially if you want many users to be able to choose among several sets of compiler options, you may 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. Rather than a user being required to specify this option on the command line for every compilation, it would automatically be in effect every time the compiler is invoked 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. Also, if you upgrade the operating system, you must change the symbolic links to the configuration file in /etc/ to point to the correct version of the configuration file.
- 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. Again, 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 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 110
- "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 will examine and process 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 then 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, then 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 in the example shown below. 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, then 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 attribute, such as **libraries** could 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 example below, the compiler will reference the **xlc** stanza in the user-defined configuration file and use 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
           options-=<B1>
xlc_replace: use=xlc
           options:=<B2>
xlc_append: use=xlc
           options+=<B3>


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

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

DEFLT:
    options=<C>
```

The stanzas in the configuration files shown above will 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** be used for any compilation invoked 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 /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. 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 11

Summary of compiler options by functional category

The XL C/C++ options available on the AIX 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 40
- "Language element control" on page 41
- "Template control (C++ only)" on page 43
- "Floating-point and integer control" on page 43
- "Error checking and debugging" on page 46
- "Listings, messages, and compiler information" on page 48
- "Optimization and tuning" on page 49
- "Object code control" on page 44
- "Linking" on page 52
- "Portability and migration" on page 53
- "Compiler customization" on page 54
- "Deprecated options" on page 55

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 79	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 80	None.	When used in conjunction with the **-E** or **-P** options, preserves or removes comments in preprocessed output.

Table 9. Compiler output options (continued)

Option name	Equivalent pragma name	Description
"-E" on page 101	None.	Preprocesses the source files named in the compiler invocation, without compiling, and writes the output to the standard output.
"-G" on page 127	None.	Generates a shared object enabled for runtime linking.
"-qmakedep, -M" on page 188	None.	Creates an output file containing targets suitable for inclusion in a description file for the **make** command.
"-MF" on page 194	None.	Specifies the target for the output generated by the **-qmakedep** or **-M** options.
"-qmkshrobj" on page 196	None.	Creates a shared object from generated object files.
"-o" on page 203	None.	Specifies a name for the output object, assembler, or executable file.
"-P" on page 213	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 241	None.	Generates an assembler language file for each source file.
"-qshowmacros" on page 245	None.	Emits macro definitions to preprocessed output.
"-qtimestamps" on page 277	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 56	None.	Compiles any file as a C++ language file.
"-qcinc (C++ only)" on page 86	None.	Places an `extern "C" { }` wrapper around the contents of include files located in a specified directory.
"-I" on page 135	None.	Adds a directory to the search path for include files.

Table 10. Compiler input options (continued)

Option name	Equivalent pragma name	Description
"-qidirfirst" on page 136	#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 139	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.
"-qsourcetype" on page 252	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 259	#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 67	None	Enables compiler support for vector data types and operators.
"-qasm" on page 71	None	Controls the interpretation of and subsequent generation of code for assembler language extensions.
"-qcpluscmt (C only)" on page 88	None.	Enables recognition of C++-style comments in C source files.
"-D" on page 92	None.	Defines a macro as in a `#define` preprocessor directive.
"-qdfp" on page 96	None.	Enables compiler support for decimal floating-point types and literals.
"-qdigraph" on page 97	#pragma options digraph	Enables recognition of digraph key combinations or keywords to represent characters not found on some keyboards.
"-qdollar" on page 98	#pragma options dollar	Allows the dollar-sign ($) symbol to be used in the names of identifiers.
"-qignprag" on page 138	#pragma options ignprag	Instructs the compiler to ignore certain pragma statements.

Table 11. Language element control options (continued)

Option name	Equivalent pragma name	Description
"-qkeyword" on page 162	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 165	► 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.
"-qlonglong" on page 186	#pragma options long long	Allows IBM `long long` integer types in your program.
"-qmacpstr" on page 187	#pragma options macpstr	Converts Pascal string literals (prefixed by the \p escape sequence) into null-terminated strings in which the first byte contains the length of the string.
"-qmbcs, -qdbcs" on page 193	#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 256	None.	Controls whether inline functions are treated as having `static` or `extern` linkage.
"-qtabsize" on page 269	None.	Sets the default tab length, for the purposes of reporting the column number in error messages.
"-qtrigraph" on page 282	None.	Enables the recognition of trigraph key combinations to represent characters not found on some keyboards.
"-U" on page 287	None.	Undefines a macro defined by the compiler or by the **-D** compiler option.
"-qutf" on page 294	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 271	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 272	None.	Specifies the maximum number of recursively instantiated template specializations that will be processed by the compiler.
"-qtemplaterecompile (C++ only)" on page 273	None.	Helps manage dependencies between compilation units that have been compiled using the **-qtemplateregistry** compiler option.
"-qtemplateregistry (C++ only)" on page 274	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 276	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 280	None.	Manages the implicit instantiation of templates.
"-qtmplparse (C++ only)" on page 281	None.	Controls whether parsing and semantic checking are applied to template definitions.

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 77	None.	Specifies whether bit fields are signed or unsigned.
"-qchars" on page 84	#pragma options chars, #pragma chars	Determines whether all variables of type `char` are treated as either `signed` or `unsigned`.
"-qenum" on page 104	#pragma options enum, #pragma enum	Specifies the amount of storage occupied by enumerations.

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

Option name	Equivalent pragma name	Description
"-qfloat" on page 114	#pragma options float	Selects different strategies for speeding up or improving the accuracy of floating-point calculations.
"-qldbl128, -qlongdouble" on page 179	#pragma options ldbl128	Increases the size of `long double` types from 64 bits to 128 bits.
"-qlonglit" on page 185	None.	In 64-bit mode, promotes literals with implicit type of `int` to `long`.
"-y" on page 306	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 58	None.	Selects either 32-bit or 64-bit compiler mode.
"-qalloca, -ma (C only)" on page 65	#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.
"-qeh (C++ only)" on page 103	None.	Controls whether exception handling is enabled in the module being compiled.
"-qexpfile" on page 108	None.	When used together with the **-qmkshrobj** or **-G** options, saves all exported symbols in a designated file.
"-qfuncsect" on page 124	#pragma options funcsect	Places instructions for each function in a separate object file control section or CSECT.
"-qinlglue" on page 147	#pragma options inlglue	When used with **-O2** or higher optimization, inlines glue code that optimizes external function calls in your application.
"-qkeepinlines (C++ only)" on page 160	None.	Keeps or discards definitions for unreferenced `extern` inline functions.
"-qpic" on page 221	None.	Generates Position-Independent Code suitable for use in shared libraries.
"-qppline" on page 222	None.	When used in conjunction with the **-E** or **-P** options, enables or disables the generation of `#line` directives.

Table 14. Object code control options (continued)

Option name	Equivalent pragma name	Description
"-qpriority (C++ only)" on page 224	#pragma options priority, #pragma priority	Specifies the priority level for the initialization of static objects.
"-qproto (C only)" on page 227	#pragma options proto	Specifies the linkage conventions for passing floating-point arguments to functions that have not been prototyped.
"-qreserved_reg" on page 234	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 235	#pragma options ro, #pragma strings	Specifies the storage type for string literals.
"-qroconst" on page 237	#pragma options roconst	Specifies the storage location for constant values.
"-qroptr" on page 238	None.	Specifies the storage location for constant pointers.
"-qrtti (C++ only)" on page 239	None.	Generates runtime type identification (RTTI) information for exception handling and for use by the typeid and dynamic_cast operators.
"-s" on page 240	None.	Strips the symbol table, line number information, and relocation information from the output file.
"-qsaveopt" on page 242	None.	Saves the command-line options used for compiling a source file, the version and level of each compiler component invoked during compilation, and other information to the corresponding object file.
"-qstatsym" on page 258	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 270	#pragma options tbtable	Controls the amount of debugging traceback information that is included in the object files.
"-qthreaded" on page 276	None.	Indicates to the compiler whether it must generate threadsafe code.
"-qtls" on page 278	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.

Table 14. Object code control options (continued)

Option name	Equivalent pragma name	Description
"-qunique (C++ only)" on page 288	None.	Generates unique names for static constructor/destructor file compilation units.
"-qweakexp" on page 302	None.	When used with the **-qmkshrobj** or **-G** options, includes or excludes global symbols marked as weak from the export list generated when you create a shared object.
"-qweaksymbol" on page 303	None.	Enables the generation of weak symbols.
"-qxcall" on page 303	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 48.

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 57	None.	Previews the compilation steps specified on the command line, without actually invoking any compiler components.
"-qcheck" on page 85	#pragma options check	Generates code that performs certain types of runtime checking.
"-qdbxextra (C only)" on page 95	#pragma options dbxextra	When used with the **-g** option, specifies that debugging information is generated for unreferenced typedef declarations, struct, union, and enum type definitions.
"-qdpcl" on page 99	None.	Generates symbols that tools based on the IBM Dynamic Probe Class Library (DPCL) can use to see the structure of an executable file.
"-qextchk" on page 109	#pragma options extchk	Generates link-time type checking information and checks for compile-time consistency.

Table 15. Error checking and debugging options (continued)

Option name	Equivalent pragma name	Description
"-qflttrap" on page 119	#pragma options flttrap	Determines the types of floating-point exception conditions to be detected at run time
"-qformat" on page 122	None.	Warns of possible problems with string input and output format specifications.
"-qfullpath" on page 124	#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.
"-g" on page 126	None.	Generates debug information for use by a symbolic debugger.
"-qhalt" on page 129	#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 130	None.	Stops compilation before producing any object, executable, or assembler source files if a specified error message is generated.
"-qheapdebug" on page 131	None.	Enables debug versions of memory management functions.
"-qinfo" on page 141	#pragma options info, #pragma info	Produces or suppresses groups of informational messages.
"-qinitauto" on page 147	#pragma options initauto	Initializes uninitialized automatic variables to a specific value, for debugging purposes.
"-qkeepparm" on page 161	None.	When used with **-O2** or higher optimization, specifies whether function parameters are stored on the stack.
"-qlinedebug" on page 181	None.	Generates only line number and source file name information for a debugger.
"-qmaxerr" on page 190	None.	Halts compilation when a specified number of errors of a specified severity level or higher is reached.
"-qoptdebug" on page 211	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 266	None.	Determines the information that appears in the symbol table.
"-qsyntaxonly (C only)" on page 267	None.	Performs syntax checking without generating an object file.

Table 15. Error checking and debugging options (continued)

Option name	Equivalent pragma name	Description
"-qwarn64" on page 301	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 46 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 74	#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 100	None.	Dumps a representation of the hierarchy and virtual function table layout of each class object to a file.
"-qflag" on page 113	#pragma options flag, ▶ C++ "#pragma report (C++ only)" on page 350	Limits the diagnostic messages to those of a specified severity level or higher.
"-qlist" on page 182	#pragma options list	Produces a compiler listing file that includes an object listing.
"-qlistopt" on page 184	None.	Produces a compiler listing file that includes all options in effect at the time of compiler invocation.
"-qphsinfo" on page 219	None.	Reports the time taken in each compilation phase to standard output.
"-qprint" on page 223	None.	Enables or suppresses listings.
"-qreport" on page 233	None.	Produces listing files that show how sections of code have been optimized.
"-qshowinc" on page 244	#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.
"-qsource" on page 251	#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.

Table 16. Listings and messages options (continued)

Option name	Equivalent pragma name	Description
"-qsrcmsg (C only)" on page 255	#pragma options srcmsg	Adds the corresponding source code lines to diagnostic messages generated by the compiler.
"-qsuppress" on page 265	None.	Prevents specific informational or warning messages from being displayed or added to the listing file, if one is generated.
"-v, -V" on page 294	None.	Reports the progress of compilation, by naming the programs being invoked and the options being specified to each program.
"-qversion" on page 296	None.	Displays the version and release of the compiler being invoked.
"-w" on page 298	None.	Suppresses informational, language-level and warning messages.
"-qxref" on page 304	#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 59	None.	Enables destructive copy operations for structures and unions.
"-qalias" on page 59	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..

Table 17. Optimization and tuning options (continued)

Option name	Equivalent pragma name	Description
"-qarch" on page 67	None.	Specifies the processor architecture for which the code (instructions) should be generated.
"-qcache" on page 81	None.	When specified with **-O4**, **-O5**, or **-qipa**, specifies the cache configuration for a specific execution machine.
"-qcompact" on page 87	#pragma options compact	Avoids optimizations that increase code size.
"-qdataimported, -qdatalocal, -qtocdata" on page 93	None.	Marks data as local or imported in 64-bit compilations.
"-qdirectstorage" on page 98	None.	Informs the compiler that a given compilation unit may reference write-through-enabled or cache-inhibited storage.
"-qenablevmx" on page 108	None.	Enables generation of vector instructions for processors that support them.
"-qfdpr" on page 112	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 132	#pragma nosimd, #pragma novector	Performs high-order loop analysis and transformations (HOT) during optimization.
"-qignerrno" on page 137	#pragma options ignerrno	Allows the compiler to perform optimizations that assume errno is not modified by system calls.
"-qipa" on page 149	None.	Enables or customizes a class of optimizations known as interprocedural analysis (IPA).
"-qisolated_call" on page 158	#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.
"-qlargepage" on page 178	None.	Takes advantage of large pages provided on POWER4™ and higher systems, for applications designed to execute in a large page memory environment.

Table 17. Optimization and tuning options (continued)

Option name	Equivalent pragma name	Description
"-qlibansi" on page 181	#pragma options libansi	Assumes that all functions with the name of an ANSI C library function are in fact the system functions.
"-qmaxmem" on page 192	#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 195	None.	Controls the generation of the table of contents (TOC), which the compiler creates for an executable file.
"-O, -qoptimize" on page 204	#pragma options optimize	Specifies whether to optimize code during compilation and, if so, at which level.
"-p, -pg, -qprofile" on page 212	None.	Prepares the object files produced by the compiler for profiling.
"-qpdf1, -qpdf2" on page 215	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 223	None.	Inserts prefetch instructions automatically where there are opportunities to improve code performance.
"-qprocimported, -qproclocal, -qprocunknown" on page 225	#pragma options procimported, #pragma options proclocal, #pragma options procunkown	Marks functions as local, imported, or unknown in 64-bit compilations.
"-Q, -qinline " on page 228	None.	Attempts to inline functions instead of generating calls to those functions, for improved performance.
"-qshowpdf" on page 246	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 247	None.	Reduces the size of the stack frame.

Table 17. Optimization and tuning options (continued)

Option name	Equivalent pragma name	Description
"-qsmp" on page 247	None.	Enables parallelization of program code.
"-qspeculateabsolutes" on page 254	None.	Works with the -qtocmerge -bl:file for non-IPA links and with the -bl:file for IPA links to disable speculation at absolute addresses.
"-qstrict" on page 260	#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 264	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.
"-qtocmerge" on page 282	None.	Enables TOC merging to reduce TOC pointer loads and improves the scheduling of external loads.
"-qtune" on page 283	#pragma options tune	Tunes instruction selection, scheduling, and other architecture-dependent performance enhancements to run best on a specific hardware architecture.
"-qunroll" on page 290	#pragma options unroll, #pragma unroll	Controls loop unrolling, for improved performance.
"-qunwind" on page 292	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
"-b" on page 75	None.	Controls how shared objects are processed by the linker.

Table 18. Linking options (continued)

Option name	Equivalent pragma name	Description
"-bmaxdata" on page 78	None.	Sets the maximum size of the area shared by the static data (both initialized and uninitialized) and the heap.
"-brtl" on page 78	None.	Controls runtime linking for the output file.
"-qcrt" on page 89	None.	Specifies whether system startup files are to be linked.
"-e" on page 101	None.	When used together with the **-qmkshrobj** or **-G** options, specifies an entry point for a shared object.
"-f" on page 110	None.	Names a file that stores a list of object files for the compiler to pass to the linker.
"-L" on page 164	None.	Searches the directory path for library files specified by the **-l** option.
"-l" on page 163	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 180	None.	Specifies whether standard system libraries and XL C/C++ libraries are to be linked.
"-R" on page 232	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 257	None.	Controls how shared and non-shared runtime libraries are linked into an application.
"-qtwolink (C++ only)" on page 286	None.	Minimizes the number of static constructors included from libraries and object files.
"-Z" on page 307	None.	Specifies a prefix for the library search path to be used by the linker.

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
"-qalign" on page 62	#pragma options align, #pragma align	Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data.

Table 19. Portability and migration options (continued)

Option name	Equivalent pragma name	Description
"-qalignrulefor (C++ only)" on page 64	None.	When used with **-qalign=power**, determines whether a 4-byte alignment ceiling is applied to non-first members of structures that are of type typedef to array of element type that exceeds the alignment ceiling.
"-qgenproto (C only)" on page 127	None.	Produces prototype declarations from K&R function definitions or function definitions with empty parentheses, and displays them to standard output.
"-qnamemangling (C++ only)" on page 198	#pragma namemangling	Chooses the name mangling scheme for external symbol names generated from C++ source code.
"-qobjmodel (C++ only)" on page 208	#pragma object_model	Sets the object model to be used for structures, unions, and classes.
"-qoldpassbyvalue (C++ only)" on page 209	#pragma pass_by_value	Specifies how classes containing const or reference members are passed in function arguments.
"-qupconv (C only)" on page 293	#pragma options upconv	Specifies whether the unsigned specification is preserved when integral promotions are performed.
"-qvecnvol" on page 295	None.	Specifies whether to use volatile or non-volatile vector registers.

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 73	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 76	None.	Determines substitute path names for XL C/C++ executables such as the compiler, assembler, linker, and preprocessor.
"-qc_stdinc (C only)" on page 90	None.	Changes the standard search location for the XL C and system header files.
"-qcpp_stdinc (C++ only)" on page 91	None.	Changes the standard search location for the XL C++ and system header files.
"-F" on page 110	None.	Names an alternative configuration file or stanza for the compiler.

Table 20. Compiler customization options (continued)

Option name	Equivalent pragma name	Description
"-qpath" on page 214	None.	Determines substitute path names for XL C/C++ executables such as the compiler, assembler, linker, and preprocessor.
"-qspill" on page 254	#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 268	None.	Applies the prefix specified by the **-B** option to the designated components.
"-W" on page 299	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 can produce unexpected results and are not guaranteed to perform as previously documented. Use with discretion.

Table 21. Deprecated options

Option name	Replacement option
-qansialias	-qalias=ansi
-qarch = 601 | 602 | 603 | pwr | pwr2 | p2sc | pwr2s | com	-qfloat=nosingle:norndsngl
-qassert	-qalias
-qfloat=emulate*	
-qfold	-qfloat=fold
-qhsflt	-qfloat=hsflt
-qhssngl	-qfloat=hssngl
-qipa=pdfname	-qpdf1=pdfname, -qpdf2=pdfname
-qmaf	-qfloat=maf
-qspnans	-qfloat=spnans
-qrrm	-qfloat=rrm

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 309

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 5, "Compiler predefined macros," on page 379

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 252

-# (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 294

-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

Usage

The **-q32** and **-q64** options override the compiler mode set by the value of the OBJECT_MODE environment variable, if it exists.

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 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
- "-qarch" on page 67
- "-qtune" on page 283

-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
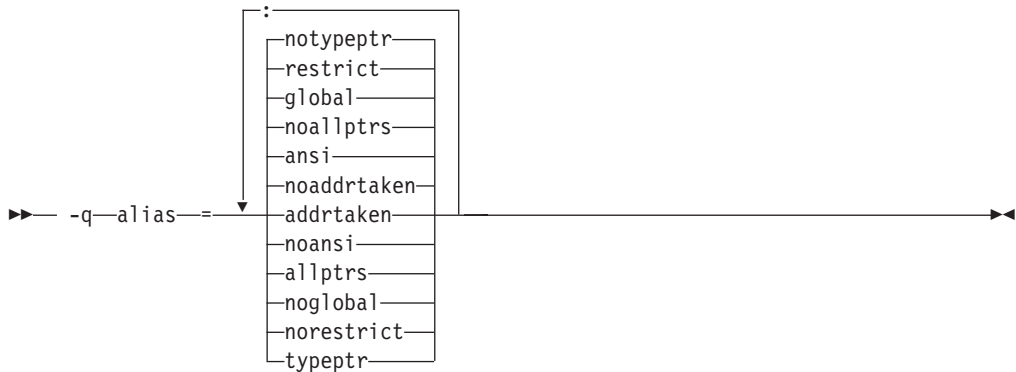
Defaults

- ► 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`.

The **-qalias=[no]ansi** option replaces the deprecated **-q[no]ansialias** option. Use **-qalias=[no]ansi** in your new applications.

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 149
- "#pragma disjoint" on page 320
- *Type-based aliasing* in the *XL C/C++ Language Reference*
- *The restrict type qualifier* in the *XL C/C++ Language Reference*

-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──=──┬─power─────┬──────────────────────────────►◄
                  ├─full──────┤
                  ├─bit_packed┤
                  ├─mac68k────┤
                  ├─natural───┤
                  ├─packed────┤
                  └─twobyte───┘
```

Pragma syntax

```
►►──#─pragma─align──(──┬─power─────┬──)──────────────────────►◄
                       ├─full──────┤
                       ├─bit_packed┤
                       ├─mac68k────┤
                       ├─natural───┤
                       ├─packed────┤
                       ├─twobyte───┤
                       └─reset─────┘
```
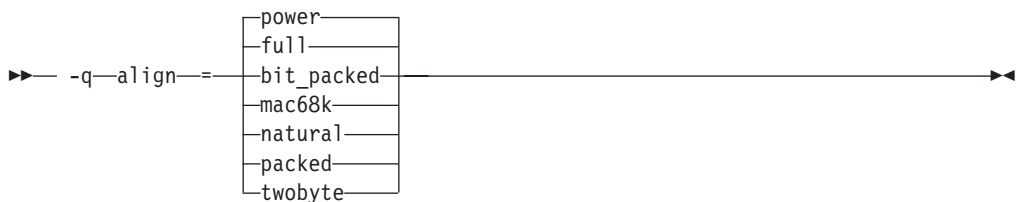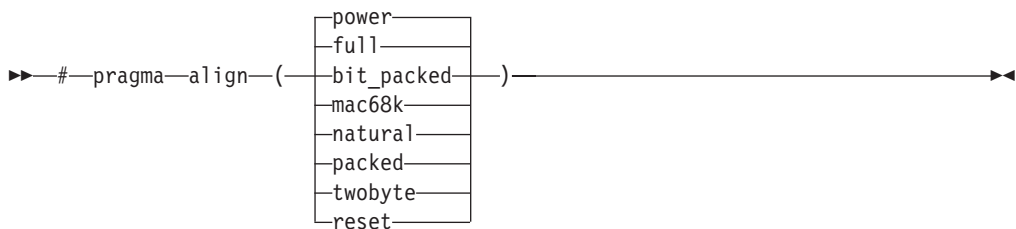
Defaults

full

Parameters

bit_packed | packed
Bit field data is packed on a bitwise basis without respect to byte boundaries.

full | power
Uses the RISC System/6000® alignment rules.

mac68k | twobyte

Uses the Macintosh alignment rules. Valid only for 32-bit compilations.

natural

Structure members are mapped to their natural boundaries. This has the same effect as the **power** suboption, except that it also applies alignment rules to `double` and `long double` members that are not the first member of a structure or union.

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 **full** suboption is the default to ensure backward-compatibility with existing objects. If backward-compatibility is not necessary, you should consider using **natural** alignment to improve potential application performance.

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=power /* Applies to B; A is unaffected  */
   struct B {
   char c;
   double d;
           } BB;   /*  BB uses power alignment rules  */
} AA;               /*  AA uses bit_packed alignment rules  /*
```

Related information
- "#pragma pack" on page 344
- "Aligning data" in the *XL C/C++ Optimization and Programming Guide*
- The __align specifier"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*

-qalignrulefor (C++ only)
Category

Portability and migration

Pragma equivalent

None

Purpose

When used with **-qalign=power**, determines whether a 4-byte alignment ceiling is applied to non-first members of structures that are of type typedef to array of element type that exceeds the alignment ceiling.

Syntax

```
►►─ -q─alignrulefor──=──power──=──┬─typedefrespectsrule───┬──────────►◄
                                  └─notypedefrespectsrule─┘
```

Defaults

-qalignrulefor=power=typedefrespectsrule

Parameters

typedefrespectsrule | notypedefrespectsrule
　　When **typedefrespectsrule** is in effect, the member follows the normal alignment rules for **-qalign=power**. This suboption provides compatibility with code compiled with **-qalign=power** with XL C++ V6.0 and earlier.

　　For XL C++ V9.0, the default is **typedefrespectsrule**.

When **notypedefrespectsrule** is in effect, a member that exceeds the alignment ceiling of 4 bytes is aligned on 4-byte boundaries. This suboption provides compatibility with code compiled with **-qalign=power** with XL C++ V7.0 and V8.0.

Predefined macros

None.

Examples

The following example uses a `typedef` declaration for an array of structures containing a member of `long long` type, which is not normally subject to a 4-byte alignment ceiling, and then uses the `typedef` as the non-first member of a structure variable declaration. The table shows the differing alignment results depending on the setting of the **-qalignrulefor=power** option.

Sample code	Alignment results	
	typedefrespectsrule	notypedefrespectsrule

`struct A {`
    `long long a1;`
    `} a;`
 	
`typedef struct A ten_A[10];`
 	
`struct B {`
    `char dummy[116];`
    `struct A ten_a[10];`
    `} b;`
 	
`struct C {`
    `char dummy[116];`
    `ten_A ten_a;`
    `} c;`
	alignment of `b.ten_a` = 8
	alignment of `b` = 8
	alignment of `c.ten_a` = 8
	alignment of `c` = 8
		alignment of `b.ten_a` = 8
		alignment of `b` = 8
		alignment of `c.ten_a` = 4
		alignment of `c` = 4

Related information
- "-qalign" on page 62

-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 92
- "__alignx" on page 438

-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 **-qarch** is set or implied to be a target architecture that supports vector processing instructions and the **-qenablevmx** compiler option is in effect. Otherwise, the compiler will ignore **-qaltivec** and issue 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:

```
xlc myprogram.c -qenablevmx -qarch=ppc64v -qaltivec
```

Related information
- "-qarch"
- "-qenablevmx" on page 108
- "-qvecnvol" on page 295
- *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─=──┬─ppc─────┬────────────────────────────────────────►◄
                ├─auto────┤
                ├─403─────┤
                ├─604─────┤
                ├─pwr3────┤
                ├─pwr4────┤
                ├─pwr5────┤
                ├─pwr5x───┤
                ├─pwr6────┤
                ├─pwr6e───┤
                ├─ppc64v──┤
                ├─ppc64───┤
                ├─ppcgr───┤
                ├─ppc64gr─┤
                ├─ppc64grsq┤
                ├─ppc970──┤
                ├─rs64a───┤
                ├─rs64b───┤
                └─rs64c───┘
```
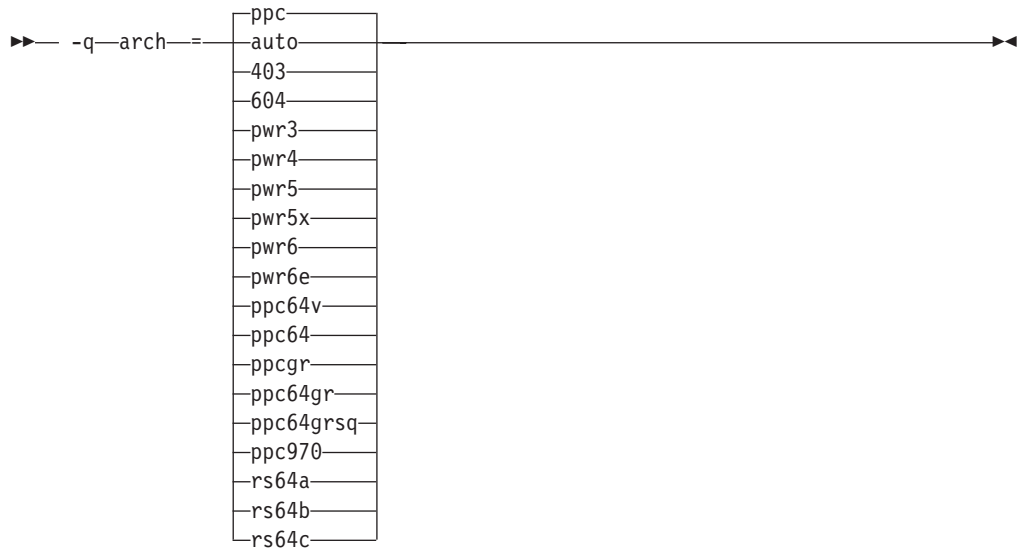
Defaults

- **-qarch=ppc** when **-q32** is in effect.
- **-qarch=ppc64** when **-q64** is in effect.
- **-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.

403
> Produces object code containing instructions that will run on the PowerPC 403™ hardware platform.

604
> Produces object code containing instructions that will run on the PowerPC 604™ hardware platform. This suboption is not valid if **-q64** is in effect.

pwr3
> Produces object code containing instructions that will run on any POWER3™, POWER4, POWER5™, POWER5+™, POWER6™, or PowerPC 970 hardware platform.

pwr4

Produces object code containing instructions that will run on the POWER4, POWER5, POWER5+, POWER6, or PowerPC 970 hardware platform.

pwr5

Produces object code containing instructions that will run on the POWER5, POWER5+, or POWER6 hardware platforms.

pwr5x

Produces object code containing instructions that will run on the POWER5+ or POWER6 hardware platforms.

pwr6

Produces object code containing instructions that will run on the POWER6 hardware platforms running in POWER6 architected mode. If you would like support for decimal floating-point instructions, be sure to specify this suboption during compilation.

pwr6e

Produces object code containing instructions that will run on the POWER6 hardware platforms running in POWER6 raw mode.

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=ppc64**.

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=ppc64gr**.

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.

rs64a

Produces object code that will run on RS64I platforms.

rs64b

Produces object code that will run on RS64II platforms.

rs64c

Produces object code that will run on RS64III platforms.

Note: The **com** suboption, and suboptions representing POWER™ and POWER2™ architectures, are no longer supported. If you would like similar floating-point behavior to that provided by this suboption, use the **-qfloat=nosingle:norndsngl** option. See "-qfloat" on page 114 for details.

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	Large page support
604	yes	no	no	no	no
rs64a	no	no	yes	no	no
rs64b	yes	yes	yes	no	no
rs64c	yes	yes	yes	no	no
pwr3	yes	yes	yes	no	no
pwr4	yes	yes	yes	no	yes
pwr5	yes	yes	yes	no	yes
pwr5x	yes	yes	yes	no	yes
ppc	no	no	no	no	yes
ppc64	no	no	yes	no	yes
ppc64gr	yes	no	yes	no	yes
ppc64grsq	yes	yes	yes	no	yes
ppc64v	yes	yes	yes	yes	yes
ppc970	yes	yes	yes	yes	yes
pwr6	yes	yes	yes	yes	yes
pwr6e	yes	yes	yes	yes	yes

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 283.

Specifying **-q64** changes the effective **-qarch** setting as follows:

Original -qarch setting	Effective setting when -q64 is specified
ppc	ppc64
ppcgr	ppc64gr

For a given application program, make sure that you specify the same **-qarch** setting when you compile each of its source files. Although the linker and loader

may detect object files that are compiled with incompatible **-qarch** settings, you should not rely on it.

Predefined macros

See "Macros related to architecture settings" on page 384 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
- "-qtune" on page 283
- "-q32, -q64" on page 58
- "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++

```
                    ┌─asm─┐
                    │     │  ┌─gcc────┐
                    │   ┌─=─┤        ├─┐
                    │   │   └─stdcpp─┘ │
►►── -q ──┬─────────┴───┴──────────────┴──────►◄
          └─noasm───────────────────────────┘
```

Defaults

- ▶ C ◀ -qasm=gcc

- ▶ C++ ◀ **-qasm=gcc** at all language levels except **compat366** or **strict98**.
  -qnoasm=stdcpp when **-qlanglvl=compat366** or **-qlanglvl=strict98** is in effect.

-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 except

compat366 | **strict98**, 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 **compat366|strict98**, and when **-qnoasm[=gcc]** is in effect. It is undefined when **-qlanglvl=compat366** | **strict98** or **-qnoasm=stdcpp** is in effect. __IBM_STDCPP_ASM is predefined to 0 when **-qlanglvl=compat366** | **strict98** or **-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 165
- "-qkeyword" on page 162
- "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 71

-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

```
►►── -q ──┬─noattr─────────────┬──────────────────────────────────────►◄
          └─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 223
- "-qxref" on page 304

-b

Category

Linking

Pragma equivalent

None.

Purpose

Controls how shared objects are processed by the linker.

Syntax

```
►►── -b──┬─dynamic─┬──────────────────────────────────────────────►◄
         ├─shared──┤
         └─static──┘
```

Defaults

-bdynamic

Parameters

dynamic | shared
> Causes the linker to process subsequent shared objects in dynamic mode. In dynamic mode, shared objects are not statically included in the output file. Instead, the shared objects are listed in the loader section of the output file.

static

Causes the linker to process subsequent shared objects in static mode. In static mode, shared objects are statically linked in the output file.

Usage

The default option, **-bdynamic**, ensures that the C library (libc) links dynamically. To avoid possible problems with unresolved linker errors when linking the C library, you must add the **-bdynamic** option to the end of any compilation sections that use the **-bstatic** option.

Predefined macros

Not applicable.

Related information
- "-brtl" on page 78

-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 268 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/usr/vacpp/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 214
- "-t" on page 268
- "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──=──┬─unsigned─┬──────────────────────────►◄
                      └─signed───┘
```

Defaults

-qbitfields=unsigned

Parameters

signed
   Bit fields are signed.

unsigned
>   Bit fields are unsigned.

Predefined macros

None.

-bmaxdata

Category

Linking

Pragma equivalent

None

Purpose

Sets the maximum size of the area shared by the static data (both initialized and uninitialized) and the heap.

Syntax

▶▶── -bmaxdata──:──*number*──◀◀

Defaults

-bmaxdata:0

Parameters

number
>   The number of bytes used representing the soft **ulimit** set by the system
>   loader. Valid values are 0 and multiples of 0x10000000 (0x10000000,
>   0x20000000, 0x30000000, ...). The maximum value allowed by the system is
>   0x80000000. If the value is 0, a single 256MB (0x10000000 byte) data segment
>   (segment 2) will be shared by the static data, the heap, and the stack. If the
>   value is non-zero, a data area of the specified size (starting in segment 3) will
>   be shared by the static data and the heap, while a separate 256 MB data
>   segment (segment 2) will be used by the stack. So, the total data size when 0 is
>   specified 0 is 256MB, and the total size when 0x10000000 is specified is 512MB,
>   with 256MB for the stack and 256MB for static data and the heap.

Predefined macros

None.

-brtl

Category

Linking

Pragma equivalent

None.

Purpose

Controls runtime linking for the output file.

Runtime linking is the ability to resolve undefined and non-deferred symbols in shared modules after the program execution has already begun. It is a mechanism for providing runtime definitions (these function definitions are not available at link-time) and symbol rebinding capabilities. Compiling with **-brtl** adds a reference to the runtime linker to your program, which will be called by your program's startup code (/lib/crt0.o) when program execution begins. Shared object input files are listed as dependents in the program loader section in the same order as they are specified on the command line. When the program execution begins, the system loader loads these shared objects so their definitions are available to the runtime linker.

Syntax

►►── -brtl ── ►◄

Usage

The main application must be built to enable runtime linking. The system loader must be able to load and resolve all symbols referenced in the main program and called modules, or the program will not execute.

DCE thread libraries and heap debug libraries are not compatible with runtime linking. Do not specify the **-brtl** compiler option if you are invoking the compiler with **xlc_r4** or **xlc++_r4**, or if the **-qheapdebug** compiler option is specified.

Predefined macros

None.

Related information
- "-b" on page 75
- "-G" on page 127

-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 101
- "-o" on page 203
- "-P" on page 213
- "-qsyntaxonly (C only)" on page 267

-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! ─┘
```
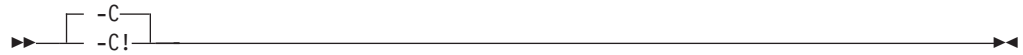
Defaults

-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 101
- "-P" on page 213

-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--|
```
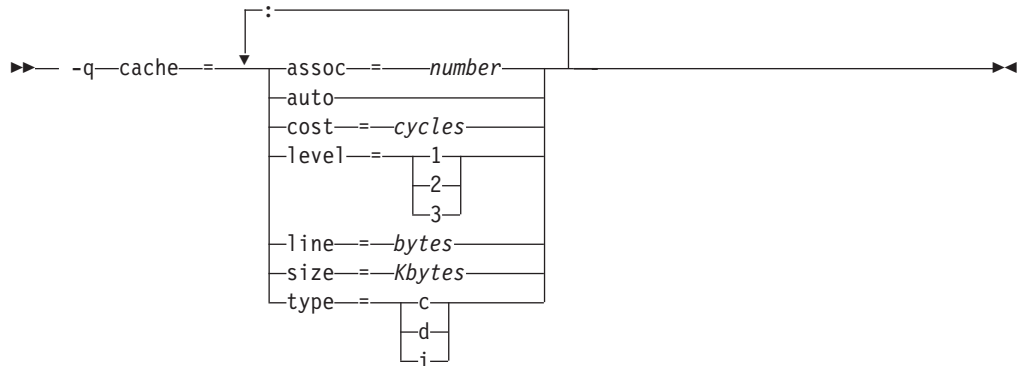
Defaults

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 81
- "-O, -qoptimize" on page 204
- "-qtune" on page 283
- "-qipa" on page 149
- "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
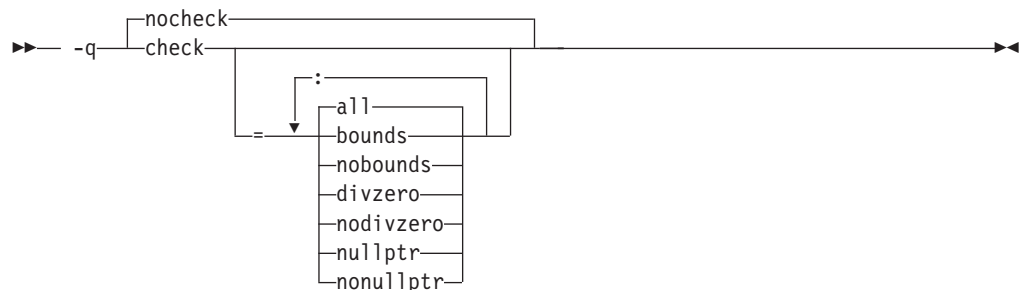
Defaults

-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();
}
```

-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

```
►►── -q──┬─nocompact─┬──────────────────────────────────────────────►◄
         └─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
```

-qcpluscmt (C only)
Category

Language element control

Pragma equivalent

None.

Purpose

Enables recognition of C++-style comments in C source files.

Syntax

```
►►── -q──┬─cpluscmt───┬────────────────────────────────────────────►◄
         └─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 //* 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 80
- "-qlanglvl" on page 165
- "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

```
            ┌─crt──┐
►►── -q──┼─nocrt─┤──────────────────────────────────────►◄
```

Defaults

-qcrt

Predefined macros

None.

Related information
- "-qlib" on page 180

-qc_stdinc (C only)
Category

Compiler customization

Pragma equivalent

None.

Purpose

Changes the standard search location for the XL C and system header files.

Syntax
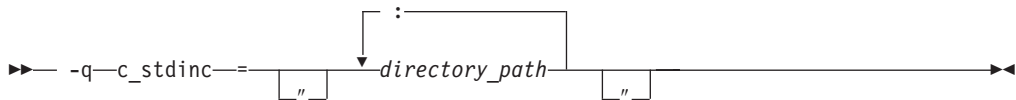
Defaults

By default, the compiler searches the directories specified in the configuration file for the XL C header files (this is normally /usr/vac/include/) and for the system header files (this is normally /usr/include/).

Parameters

directory_path
> The path for the directory where the compiler should search for the XL C and system 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 and system headers, you use a configuration file to do so; see "Directory search sequence for include files" on page 13 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
- "-qstdinc" on page 259
- "-qinclude" on page 139
- "Directory search sequence for include files" on page 13
- "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++ and system header files.

Syntax

Defaults

By default, the compiler searches the directories specified in the configuration file for the XL C++ header files (this is normally /usr/vacpp/include/) and for the system header files (this is normally /usr/include/).

Parameters

directory_path
> The path for the directory where the compiler should search for the XL C++ and system 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++ and system headers, you use a configuration file to do so; see "Directory search sequence for include files" on page 13 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
- "-qstdinc" on page 259
- "-qinclude" on page 139
- "Directory search sequence for include files" on page 13
- "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.

To aid in program portability and standards compliance, the operating system
provides several header files that refer to macro names you can set with the **-D**
option. You can find most of these header files either in the /usr/include directory
or in the /usr/include/sys directory. To ensure that the correct macros for your
source file are defined, use the **-D** option with the appropriate macro name. For
example, if your source file includes the /usr/include/sys/stat.h header file, you
must compile with the option -D_POSIX_SOURCE to pick up the correct
definitions for that file.

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

AIX 4.2 and later provides support for files greater than 2 gigabytes in size so you
can store large quantities of data in a single file. To allow large file manipulation in
your application, compile with the -D_LARGE_FILES and **-qlonglong** compiler
options. For example:
```
xlc myprogram.c -D_LARGE_FILES -qlonglong
```

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 287
- Chapter 5, "Compiler predefined macros," on page 379
- "Header files" in the *AIX Files Reference*

-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 ──┘
```
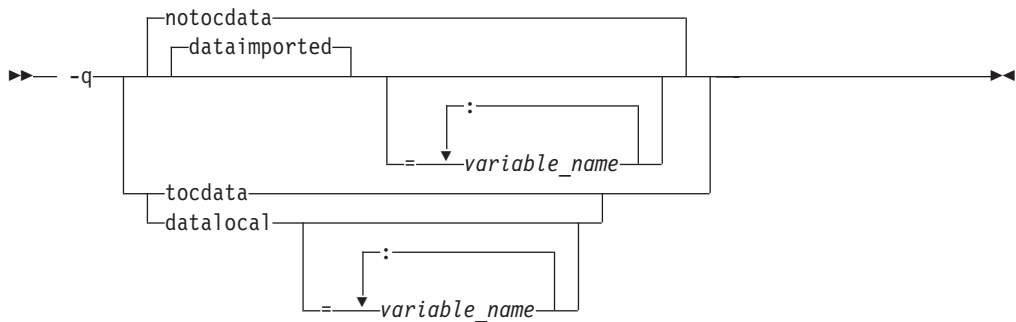
Defaults

-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. You can also use can the **c++filt** utility provided by the compiler for a side-by-side listing of source names and mangled names; see "Demangling compiled C++ names" in the *XL C/C++ Optimization and Programming Guide* for details. (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, 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.

Related information
- "-qprocimported, -qproclocal, -qprocunknown" on page 225

-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 124
- "-qlinedebug" on page 181
- "-g" on page 126
- "#pragma options" on page 340
- "-qsymtab (C only)" on page 266

-qdfp

Category

Language element control

Pragma equivalent

None.

Purpose

Enables compiler support for decimal floating-point types and literals.

Syntax

```
►►──-q──┬─nodfp─┬──────────────────────────────────────────────►◄
        └─dfp───┘
```

Defaults

-qnodfp

Usage

If you enable **-qdfp** for a **-qarch** value that does not support decimal floating-point instructions, **-qfloat=dfpemulate** is automatically enabled, and the decimal floating-point operations are performed by software. This may cause a slowdown in the application's runtime performance.

Note that runtime support for decimal floating-point operations is available only on AIX for POWER version 5.3 with the 5300-06 Technology Level or later. If you enable **-qdfp** on a version of the operating system that does not provide runtime support, your application will compile, but it may not link or run.

Programs that use decimal floating-point functions or macros defined in the <math.h> include file should not be compiled on AIX 5.2 or on older levels of AIX 5.3 or 5.4. because those functions and macros won't be retrofitted into 5.2.

Predefined macros

When **-qdfp** is in effect, __IBM_DFP__ is predefined to a value of 1; otherwise it is undefined.

Related information
- "-qarch" on page 67
- "-qfloat" on page 114

-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 165
- "-qtrigraph" on page 282

-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

```
►►── -q──┬─nodirectstorage─┬──────────────────────────────────►◄
         └─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

```
►►── -q──┬─nodollar─┬──────────────────────────────────────────►◄
         └─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 165

-qdpcl

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Generates symbols that tools based on the IBM Dynamic Probe Class Library (DPCL) can use to see the structure of an executable file.

DPCL is an open-source set of libraries used by application performance analysis tools (for more information, visit http://dpcl.sourceforge.net). When **-qdpcl** is in effect, the compiler emits symbols to define blocks of code in a program; you can then use tools that use the DPCL interface to examine performance information such as memory usage for object files compiled with this option.

Syntax

```
►►── -q──┬─nodpcl─┬──────────────────────────────────────────►◄
          └─dpcl───┘
```

Defaults

-qnodpcl

Usage

You must specify **-qdpcl** together with the **-g** option to ensure that the compiler generates debugging information required by debugging and program analysis tools.

-qdpcl is not supported for any optimization level except zero. If a non-zero optimization level is specified or implied by other options, **-qdpcl** will be disabled.

You cannot specify the **-qipa** or **-qsmp** options together with **-qdpcl**.

Predefined macros

None.

Related information
- "-g" on page 126
- "-qipa" on page 149
- "-qsmp" on page 247

-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** or **-G** options, specifies an entry point for a shared object.

Syntax

```
▶▶── -e ──┬─noentry─┬──────────────────────────────────────── ▶◀
          └─ name ──┘
```

Defaults

-e=noentry

Parameters

name
    The name of the entry point for the shared executable.

Usage

When linking object files, it is recommended that you *do not* use the **-e** option. The default entry point of the executable output is __start. Changing this label with the **-e** flag can cause erratic results.

This option is used only together with the **-qmkshrobj** or **-G** option. See the description for the "-qmkshrobj" on page 196 for more information.

Predefined macros

None.

Related information
- "-qmkshrobj" on page 196
- "-G" on page 127

-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
- "-qppline" on page 222
- "-C, -C!" on page 80
- "-P" on page 213
- "-qsyntaxonly (C only)" on page 267

-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

Defaults

-qeh=v6

Parameters

v6 Generates exception handling code, compatible with VisualAge® C++ V6.0, that correctly handles try-catch blocks nested within other catch blocks.

v5 Generate exception handling code that is compatible with VisualAge C++ V5.0.

Specifying **-qeh** with no suboption is equivalent to **-qeh=v6**.

Predefined macros

__EXCEPTIONS is predefined to 1 when **-qeh** is in effect; otherwise, it is undefined.

Related information
- "-qrtti (C++ only)" on page 239

-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───┘
```
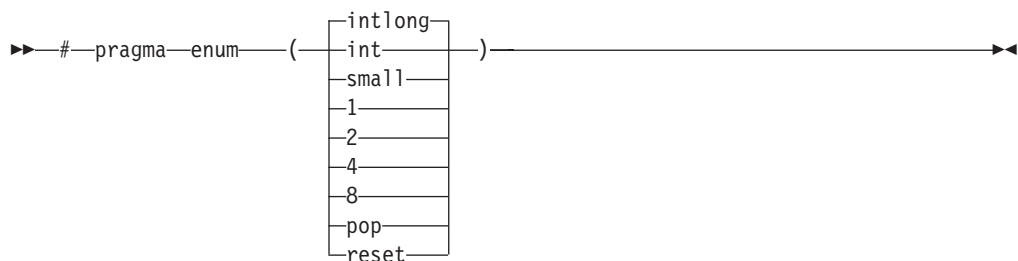
Defaults

-qenum=intlong

Parameters

1 Specifies that enumerations occupy 1 byte of storage, are of type `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 | int
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.

intlong
Specifies that enumerations occupy 8 bytes of storage if the range of values in the enumeration exceeds the limit for `int`. If the range of values in the enumeration does not exceed the limit for `int`, the enumeration will occupy 4 bytes of storage and is represented by `int`.

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. Signage 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	char	int	short	int	int	int	long long	long long	long	long
-128..127	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) ..2147483647	ERROR[1]	int	ERROR[1]	int	int	int	long long	long long	long	long

Table 23. Enumeration sizes and types (continued)

0..4294967295	ERROR[1]	unsigned int	ERROR[1]	unsigned int	unsigned int	unsigned int	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]
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	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int
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]	long[2]	long[2]	unsigned long long[2]	unsigned long long[2]	unsigned long[2]	unsigned long[2]

Note:

1. These enumerations are too large for the **-qenum=1|2|48** settings. 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**.

2. ▶ C ◀ Enumeration types must not exceed the range of int when compiling C applications to ISO C 1989 and ISO C 1999 Standards. When the **stdc89** |

stdc99 language level in effect, the compiler will behave as follows if the value of an enumeration exceeds the range of int:

a. If **-qenum=int** is in effect, a severe error message is issued and compilation stops.

b. 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 /* second reset isn't required */
```

The range of enum constants must fall within the range of either unsigned int or int (signed int). For example, the following code segments contain errors:

```
#pragma enum=small
enum e_tag { a=-1,
            b=2147483648   /* error: larger than maximum int */
          } e_var;
#pragma options enum=reset
```

Predefined macros

None.

-qenablevmx

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Enables generation 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

```
                 ┌─noenablevmx─┐
►►── -q───┴─enablevmx───┴──────────────────────────────────────────►◄
```

Defaults

-qnoenablevmx

Usage

Do not specify **-qenablevmx** unless your operating system version supports vector instructions.

If **-qnoenablevmx** is in effect, **-qaltivec**, **-qvecnvol**, and **-qhot=simd** cannot be used.

Predefined macros

None.

Related information
- "-qaltivec" on page 67
- "-qarch" on page 67
- "-qhot" on page 132
- "-qvecnvol" on page 295

-qexpfile

Category

Object code control

Pragma equivalent

None.

Purpose

When used together with the **-qmkshrobj** or **-G** options, saves all exported symbols in a designated file.

Syntax

```
►►── -q─expfile──=──filename───────────────────────────────────────►◄
```

Parameters

file_name
    The name of the file to which exported symbols are written.

Usage

This option is only valid when used with the **-qmkshrobj** or **-G** options.

Predefined macros

None.

Related information
- "-qmkshrobj" on page 196
- "-G" on page 127

-qextchk

Category

Error checking and debugging

Pragma equivalent

#pragma options [no]extchk

Purpose

Generates link-time type checking information and checks for compile-time consistency.

Syntax

```
          ┌─noextchk─┐
►►── -q────┴─extchk───┴──────────────────────────────────────────────►◄
```

Defaults

-qnoextchk

Usage

This option does not perform type checking on functions or objects that contain references to incomplete types.

Predefined macros

None.

Examples

To compile `myprogram.c` so that link-time checking information is produced, enter:

```
xlc myprogram.c -qextchk
```

-f

Category

Linking

Pragma equivalent

None.

Purpose

Names a file that stores a list of object files for the compiler to pass to the linker.

Syntax

▶▶── -f─*filelistname*──▶◀

Usage

The *filelistname* file should contain only the names of object files. There should be one object file per line.

This option is the same as the **-f** option for the **ld** command.

Predefined macros

None.

Examples

To pass the list of files contained in `myobjlistfile` to the linker, enter:

```
xlc -f/usr/tmp/myobjlistfile
```

-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 supplied 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 76
- "-t" on page 268
- "-W" on page 299

-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

```
►►── -q──┬─nofdpr─┬──────────────────────────────────────────►◄
         └─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 215

-qflag

Category

Listings, messages, and compiler information

Pragma equivalent

#pragma options flag, "#pragma report (C++ only)" on page 350

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
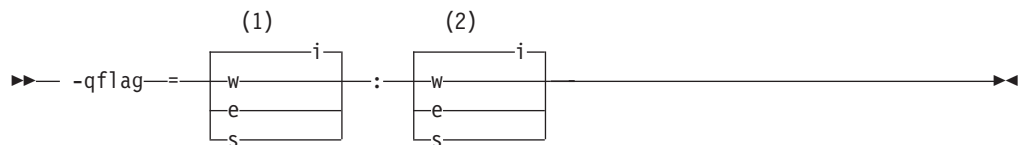
```
                        (1)                   (2)
                         i                     i
►►── -qflag──=──┬──w──┬──:──┬──w──┬──────────────────────►◄
               ├──e──┤      ├──e──┤
               └──s──┘      └──s──┘
```

Notes:

1 Minimum severity level of messages reported in listing

2 Minimum severity level of messages reported on terminal

-qflag syntax – C++
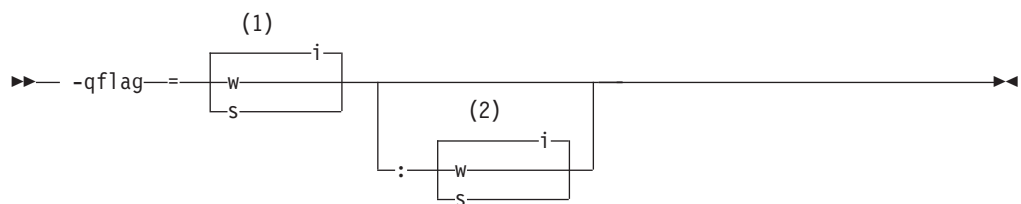
```
                      (1)
                       i
►►── -qflag──=──┬──w──┬─────────────────────────────────►◄
               └──s──┘
                         (2)
                          i
                └──:──┬──w──┬──┘
                      └──s──┘
```

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 141
- "-w" on page 298
- "Compiler messages" on page 17

-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

Option:

```
                                    :
                              ┌──nospnans──┐
                              ├──single────┤
                              ├──norsqrt───┤
                              ├──norrm─────┤
                              ├──rngchk────┤
                              ├──rndsngl───┤
                              ├──norelax───┤
                              ├──nonans────┤
                              ├──maf───────┤
                              ├──nohssngl──┤
                              ├──nohsflt───┤
                              ├──nohscmplx─┤
                              ├──fold──────┤
                              ├──nofltint──┤
                              ├──nofenv────┤
                              ├──dfpemulate┤
►►── -q─float──=──┬───────────┼──nodfpemulate──────────────────────────►◄
                              ├──fenv──────┤
                              ├──fltint────┤
                              ├──nofold────┤
                              ├──hscmplx───┤
                              ├──hsflt─────┤
                              ├──hssngl────┤
                              ├──nomaf─────┤
                              ├──nans──────┤
                              ├──relax─────┤
                              ├──norndsngl─┤
                              ├──norngchk──┤
                              ├──rrm───────┤
                              ├──rsqrt─────┤
                              ├──nosingle──┤
                              └──spnans────┘
```
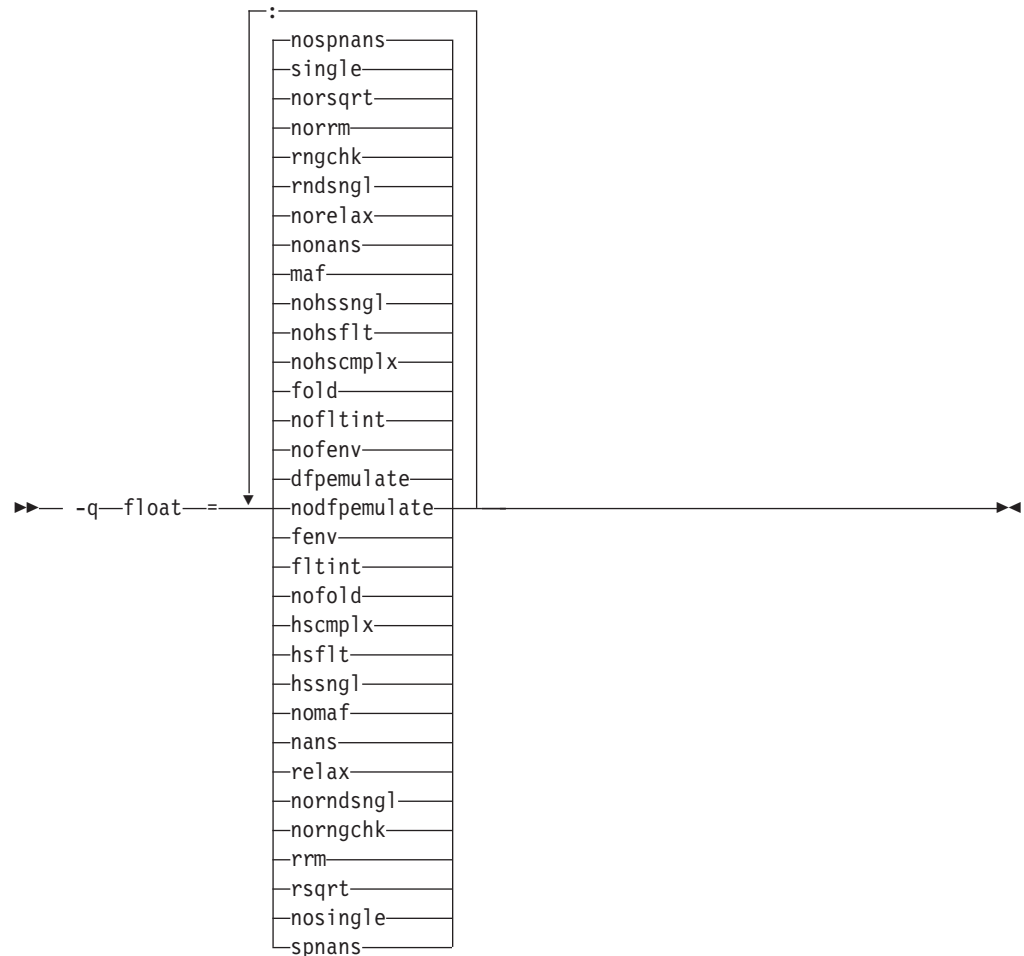
Defaults

- **-qfloat=dfpemulate:nofenv:nofltint:fold:
  nohscmplx:nohsflt:nohssnglmaf:nonans:norelax:rndsngl:rngchk:norrm:norsqrt:single:nospnan**
- **-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.
- **-qfloat=nodfpemulate** when **-qarch=pwr6** is in effect.

Parameters

dfpemulate | nodfpemulate

Specifies whether decimal floating-point computations are implemented in
hardware instructions or emulated in software by calls to library functions.
nodfpemulate is only valid on a system that supports decimal floating-point
instructions; that is, on AIX 5.3 and higher, and with **-qarch=pwr6** in effect.
nodfpemulate is the recommended setting for those systems, and results in

improved performance of decimal floating-point operations and overall program runtime performance. **dfpemulate** is required for all other **-qarch** values.

Note that **-qdfp** must also be enabled for either suboption to have any effect.

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 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**.

The **-qfloat=[no]fold** option replaces the deprecated **-q[no]fold** option. Use **-qfloat=[no]fold** in your new applications.

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**.

hssngl | nohssngl

Specifies that single-precision expressions are rounded only when the results are stored into memory locations, but not after expression evaluation. Using **hssngl** can improve runtime performance and is safer than using **hsflt**.

This option only affects double-precision (double) expressions cast to single-precision (float) and used in an assignment operator for which a store instruction is generated, when **-qfloat=nosingle** is in effect. Do not use this option if you are compiling with the default **-qfloat=single**.

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.

The **-qfloat=[no]maf** option replaces the deprecated **-q[no]maf** option. Use **-qfloat=[no]maf** in your new applications.

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.

The **hsflt** option overrides the **nans** option.

The **-qfloat=[no]nans** option replaces the deprecated **-qfloat=[no]spnans** option and the **-q[no]spnans** option. Use **-qfloat=[no]nans** in your new applications.

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.

norndsngl | rndsngl

Rounds the result of each single-precision operation to single-precision, rather than waiting until the full expression is evaluated. It sacrifices speed for consistency with results from similar calculations on other types of computers.

This option only affects double-precision expressions cast to single-precision. You can only specify **norndsngl** when **-qfloat=nosingle** is in effect.

The **hsflt** suboption overrides the **rndsngl** option.

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.

The **-qfloat=[no]rrm** option replaces the deprecated **-q[no]rrm** option. Use **-qfloat=[no]rrm** in your new applications.

rsqrt | norsqrt
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**.

single | nosingle
Allows single-precision arithmetic instructions to be generated for single-precision floating-point values. All PowerPC processors support single-precision instructions; however, if you wish to preserve the behavior of applications compiled for earlier architectures, in which all floating-point arithmetic was performed in double-precision and then truncated to single-precision, you can use **-qfloat=nosingle:norndsngl**. This suboption provides computation precision results compatible with those provided by the deprecated options **-qarch=com|pwr|pwrx|pwr2|p2sc|601|602|603**. **-qfloat=nosingle** can be specified in 32-bit mode only.

spnans | nospnans
Generates extra instructions to detect signalling NaN on conversion from single-precision to double-precision.

The **hsflt** suboption overrides the **spnans** suboption.

Note: As of the V9.0 release of the compiler, the **emulate|noemulate** suboptions are deprecated.

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

__IBM_DFP_SW_EMULATION__ is predefined to a value of 1 when **-qfloat=dfpemulate** is in effect; otherwise it is undefined.

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 67
- "-qflttrap"
- "-qldbl128, -qlongdouble" on page 179
- "-qstrict" on page 260

-qflttrap

Category

Error checking and debugging

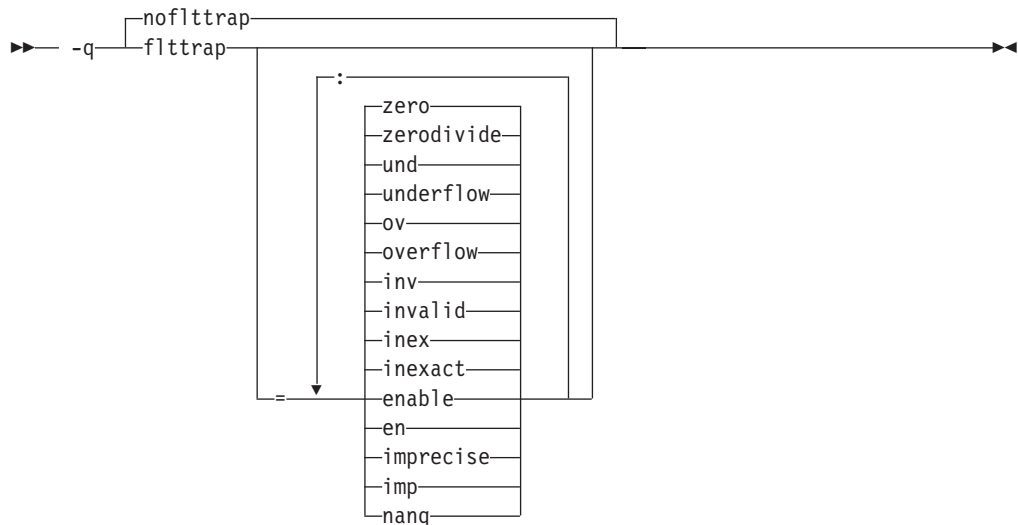### Pragma equivalent

#pragma options [no]flttrap

Purpose

Determines the types of floating-point exception conditions to be detected at run time

Syntax

```
         ┌─noflttrap──────────────────────────────────┐
►►── -q──┤                                             ├──►◄
         └─flttrap─┬──────────────────────────────────┤
                   │              ┌─:──────────┐       │
                   │              │ ┌─zero─────┐│       │
                   │              │ ├─zerodivide┤       │
                   │              │ ├─und───────┤       │
                   │              │ ├─underflow─┤       │
                   │              │ ├─ov────────┤       │
                   │              │ ├─overflow──┤       │
                   │              │ ├─inv───────┤       │
                   │              │ ├─invalid───┤       │
                   │              ▼ ├─inex──────┤       │
                   └─=────────────┴─┼─inexact───┼───────┘
                                    ├─enable────┤
                                    ├─en────────┤
                                    ├─imprecise─┤
                                    ├─imp───────┤
                                    └─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. If **imprecise** is in effect, traps will not report exactly where the exception occurred.

imprecise, imp
> Enables imprecise detection of the specified exceptions. The compiler generates instructions after a block of code and just before the `main` program returns, to check if any of the specified exceptions (**overflow**, **underflow**, **zerodivide**, **invalid**, or **inexact**) have occurred. If an exception has occurred, an exception status flag is set in the Floating-Point Status and Control Register, but the exact location of the exception is not determined. Because instructions are not generated after each floating-point operation and function call to check for exceptions, this suboption can result in a slight performance improvement.

inexact, inex
> Enables the detection of floating-point inexact operations. If **imprecise** is not also specified, the compiler generates instructions after each floating-point operation and function call to check if an inexact operation exception has occurred. 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 **imprecise** is not also specified, the compiler generates instructions after each floating-point operation and function call to check if an invalid operation exception has occurred. 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 **imprecise** is not also specified, the compiler generates instructions after each floating-point operation and function call to check if an overflow exception has occurred. 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 **imprecise** is not also specified, the compiler generates instructions after each floating-point operation and function call to check if an underflow exception has occurred. 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 **imprecise** is not also specified, the compiler generates instructions after each floating-point operation and function call to check if a zero-divide exception has occurred. 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.

If you use **-qflttrap=inv:en** to compile a program containing an IEEE invalid SQRT operation and you specify a **-qarch** target that does not implement the `sqrt` instruction set, the expected SIGTRAP signal will not occur when you run the program. You can fix this problem by specifying the following command before running the program:

```
export SQRT_EXCEPTION=3.1
```

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 114
- "-qarch" on page 67

-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

```
►►─── -q─┬─noformat───────────────────────────────┬───────────────►◄
         └─format─┬──────────────────────────────┬┘
                  │        ┌─── : ───────┐        │
                  └─ = ─▼──┬─all────┬────┘
                           ├─noall──┤
                           ├─exarg──┤
                           ├─noexarg┤
                           ├─nlt────┤
                           ├─nonlt──┤
                           ├─sec────┤
                           ├─nosec──┤
                           ├─y2k────┤
                           ├─noy2k──┤
                           ├─zln────┤
                           └─nozln──┘
```
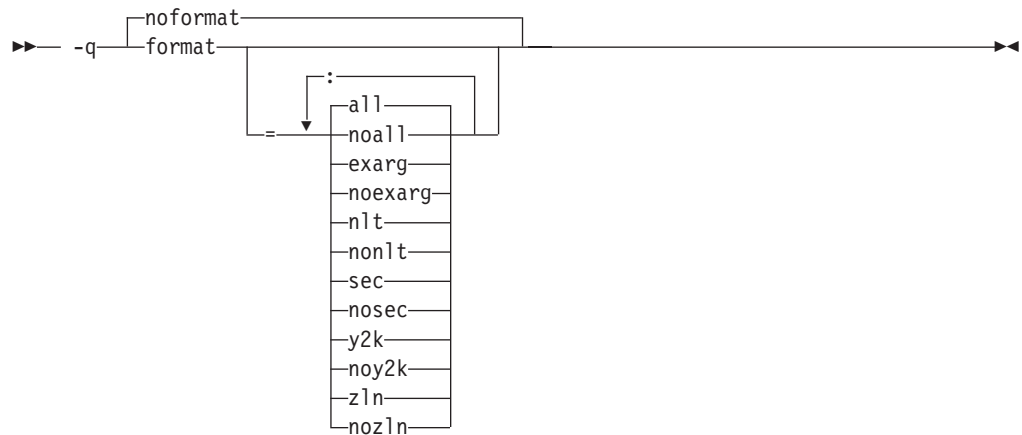
Defaults

-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 181
- "-g" on page 126

-qfuncsect

Category

Object code control

Pragma equivalent
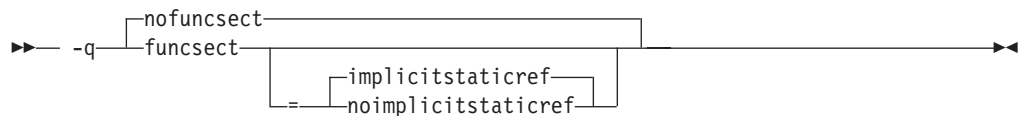
#pragma options [no]funcsect

Purpose

Places instructions for each function in a separate object file control section or CSECT.

When **-qfuncsect** is specified the compiler generates references from each function to the static data area, if one exists, in order to ensure that if any function from that object file is included in the final executable, the static data area also is included. This is done to ensure that any static strings or strings from a pragma comment, possible containing copyright information, are also included in the executable. This can, in some cases, cause code bloat or unresolved symbols at link time.

When **-qnofuncsect** is in effect, each object file consists of a single control section combining all functions defined in the corresponding source file. You can use **-qfuncsect** to place each function in a separate control section.

Syntax

```
              ┌─nofuncsect─────────────────────────────┐
►►── -q──┴─funcsect───────────────────────────────┴──────────────────────►◄
                      │        ┌─implicitstaticref───┐   │
                      └─=─┴─noimplicitstaticref─┘
```

Defaults

-qnofuncsect

Parameters

implicitstaticref | **noimplicitstaticref**
> Specifies whether references to the static data section of the object file by functions contained in static variables, virtual function tables, or exception handling tables, are maintained.
>
> When your code contains a **#pragma comment** directive or a static string for copyright information purposes, the compiler automatically places these strings in the static data area, and generates references to these static data areas in the object code.
>
> When **implicitstaticref** is in effect, any references to the static area by functions that are removed by the linker's garbage collection procedures are maintained; this may result in unresolved function definition errors by the linker.
>
> When **noimplicitstaticref** is in effect, these references to the static area are removed, allowing for successful linking and potentially reduced executable size; note, however, that this may result in a failure to include the static data area and any copyright information that it may contain.

Specifying **-qfuncsect** with no suboption implies **implicitstaticref**.

Usage

Using multiple control sections increases the size of the object file, but can reduce the size of the final executable by allowing the linker to remove functions that are not called or that have been inlined by the optimizer at all places they are called.

The pragma directive must be specified before the first statement in the compilation unit.

Predefined macros

None.

Related information
- "#pragma comment" on page 318
- "-qtwolink (C++ only)" on page 286

-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

- "-qdbxextra (C only)" on page 95
- "-qfullpath" on page 124
- "-qlinedebug" on page 181
- "-O, -qoptimize" on page 204
- "-qsymtab (C only)" on page 266

-G

Pragma equivalent

None.

Purpose

Generates a shared object enabled for runtime linking.

Syntax

►►── -G ──►◄

Usage

The compiler will automatically export all global symbols from the shared object unless you specify which symbols to export by using **-bE:**, **-bexport:**, or **-bnoexpall**. You can also prevent weak symbols from being exported by using the **-qnoweakexp** option. To save the export list to a file, use the **-qexpfile** option.

Predefined macros

None.

Related information

- "-b" on page 75
- "-brtl" on page 78
- "-qexpfile" on page 108
- "-qmkshrobj" on page 196
- "-qweakexp" on page 302
- Options that control linking: Options for linker output control
- "Shared Objects and Runtime Linking" in *AIX General Programming Concepts: Writing and Debugging Programs*
- **ld** in *AIX Commands Reference, Volume 3: i through m*

-qgenproto (C only)

Category
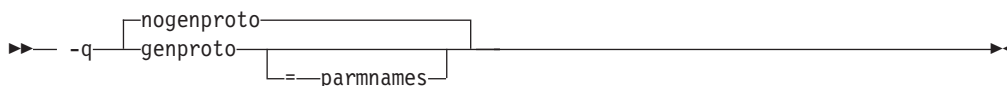
Portability and migration

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).

s ▶ 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 19.

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 113
- "-qmaxerr" on page 190

-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:
    `15dd-number`

where:

dd   Is the two-digit code representing the compiler component that produces the message. See "Compiler message format" on page 18 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 17

-qheapdebug
Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Enables debug versions of memory management functions.

The compiler ships a set of "debug" versions of the standard memory management functions defined in stdlib.h (such as _debug_calloc and _debug_malloc); the header files for these functions are found in the product include directory (usr/vacpp/include). By default, the compiler uses the regular memory management functions (such as `calloc` and `malloc`) and does not preinitialize their local storage. When **-qheapdebug** is in effect, the compiler searches for header files first in the product include directory, where the debug versions of memory management functions are stored, and then in the system include directory.

Syntax

```
              ┌─noheapdebug─┐
►►─ -q──┴─heapdebug───┴─────────────────────────►◄
```

Defaults

-qnoheapdebug

Usage

For complete information on the debug memory management functions, see "Memory debug library functions" in the *XL C/C++ Optimization and Programming Guide*.

Predefined macros

__DEBUG_ALLOC__ is defined to 1 when **-qheapdebug** is in effect; otherwise, it is undefined.

Examples

To compile myprogram.c with the debug versions of memory management functions, enter:

```
xlc -qheapdebug myprogram.c -o testing
```

Related information
*   "Debugging memory heaps" in the *XL C/C++ Optimization and Programming Guide*

-qhot

Category

Optimization and tuning

Pragma equivalent

#pragma novector, #pragma nosimd

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.
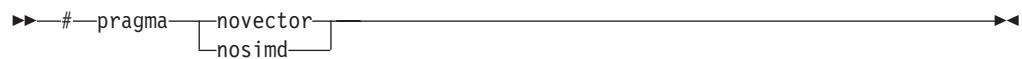
You can use the pragma directives to disable these transformations for selected sections of code.

Syntax

Option syntax

```
          ┌─nohot─────────────────────────────────────┐
►►──-q──┬────────────────────────────────────────────────┬──────►◄
        └─hot─┬──────────────────────────────────────┘
              │          ┌─:────────────────────────┐
              │          ▼                          │
              └─=──┬──────┬─noarraypad───────────────┤
                   │      ├─arraypad─────────────────┤
                   │      │         └─=──number─┘     │
                   │      │              ┌─1─┐        │
                   │      ├─level──=──┴─0─┴───────────┤
                   │      ├─simd─────────────────────┤
                   │      ├─nosimd───────────────────┤
                   │      ├─vector───────────────────┤
                   │      └─novector─────────────────┘
```

Pragma syntax

```
►►──#──pragma──┬─novector─┬──────────────────────────────────────►◄
               └─nosimd───┘
```
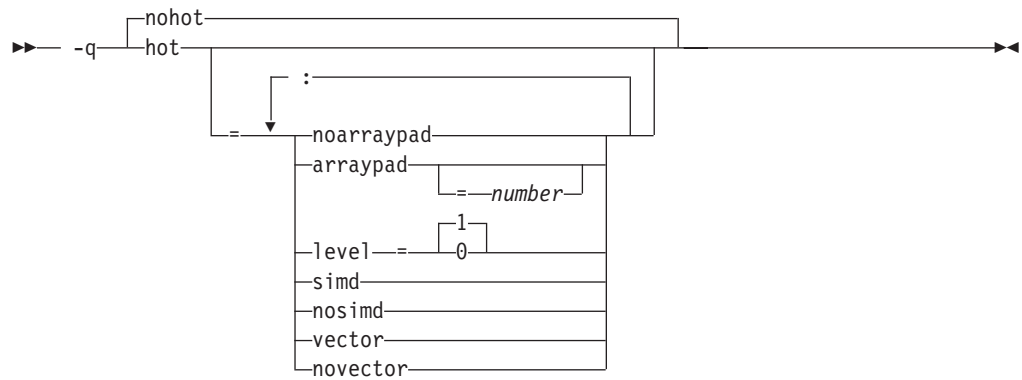
Defaults

- -qnohot
- **-qhot=noarraypad:level=0:nosimd:vector** when **-O3** is in effect.
- **-qhot=noarraypad:level=1:nosimd:vector** when **-qsmp**, **-O4** or **-O5** is in effect.
- Specifying **-qhot** without suboptions is equivalent to **-qhot=noarraypad:level=1:nosimd:vector**. If you specify **-qhot** without suboptions with **-qenablevmx** and a **-qarch** value that supports vector processing, **-qhot=simd** is enabled by default.

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:nosimd:noarraypad.

level=1 (option only)
    Performs the default set of high-order transformations.

simd (option only) | nosimd
    When simd is in effect, the compiler converts certain operations that are
    performed in a loop on successive elements of an array into a call to a vector
    instruction. This call calculates several results at one time, which is faster than
    calculating each result sequentially. Applying this suboption is useful for
    applications with significant image processing demands.

    This suboption has effect only if you specify an architecture that supports
    vector processing, and **-qenablevmx** is enabled.

    Note: This suboption is supported on AIX 5L™ Version 5.3 with the 5300-03
    Recommended Maintenance package and higher.

    nosimd disables the conversion of loop array operations into calls to vector
    instructions.

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. If the operations are in a
    loop, the vector version of the routine is called. If the operations are scalar, the
    scalar version of the routine is called. 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.

Usage

If you do not also specify an optimization level when specifying **-qhot** on the
command line, the compiler assumes **-O2**.

If you specify **-O3**, the compiler assumes **-qhot=level=0**; to prevent all HOT
optimizations with **-O3**, you must specify **-qnohot**.

If you want to override the default **level** setting of **1** when using **-qsmp**, **-O4** or
-O5, be sure to specify **-qhot=level=0** *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 also use the **-qreport** option in conjunction with **-qhot** to produce a
pseudo-C report showing how the loops were transformed; see "-qreport" on page
233 for details.

Predefined macros

None.

Examples

The following example shows the usage of **#pragma nosimd** to disable **-qhot=simd** for a specific `for` loop:

```
...
#pragma nosimd
for (i=1; i<1000; i++) {
    /* program code */
}

...
```

Related information
- "-qarch" on page 67
- "-qenablevmx" on page 108
- "-O, -qoptimize" on page 204
- "-qstrict" on page 260
- "-qsmp" on page 247
- *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 13 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 259
- "-qinclude" on page 139
- "Directory search sequence for include files" on page 13
- "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

```
        ┌─noidirfirst─┐
►►── -q─┴─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 13.) 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 135
- "-qinclude" on page 139
- "-qstdinc" on page 259
- "-qc_stdinc (C only)" on page 90
- "-qcpp_stdinc (C++ only)" on page 91
- "Directory search sequence for include files" on page 13

-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 204

-qignprag

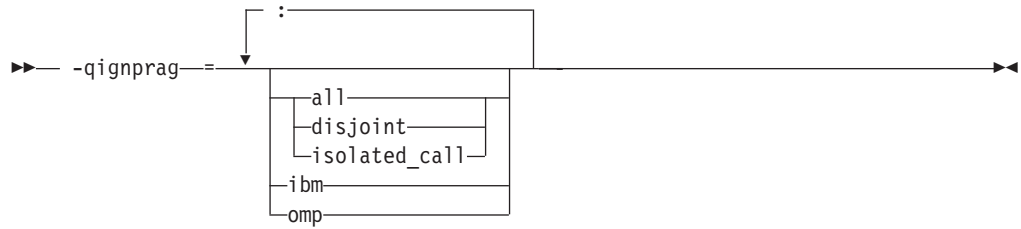### Category

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

```
>>-- -qignprag--=--+-----------------------------+----------------><
                   |        .-:------------------.          |
                   |        v                    |          |
                   +--------+--all------------+--+          |
                   |        +--disjoint--------+  |         |
                   |        '--isolated_call---'  |         |
                   +--ibm----------------------+  |
                   '--omp----------------------'
```

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 and all IBM SMP directives (such as **#pragma ibm parallel_loop** and **#pragma ibm schedule**) 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 320
- "-qisolated_call" on page 158
- "#pragma ibm snapshot" on page 327
- "Pragma directives for parallel processing" on page 359

-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 13.

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 13

-qinfo

Category

Error checking and debugging

Pragma equivalent
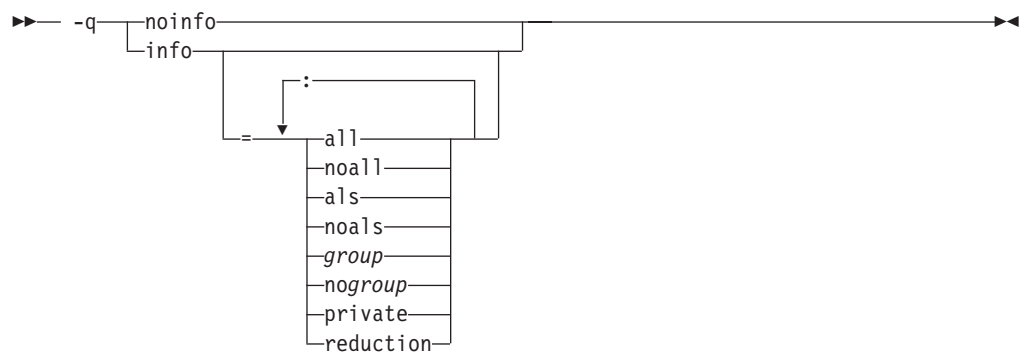
#pragma options [no]info, #pragma info

Purpose
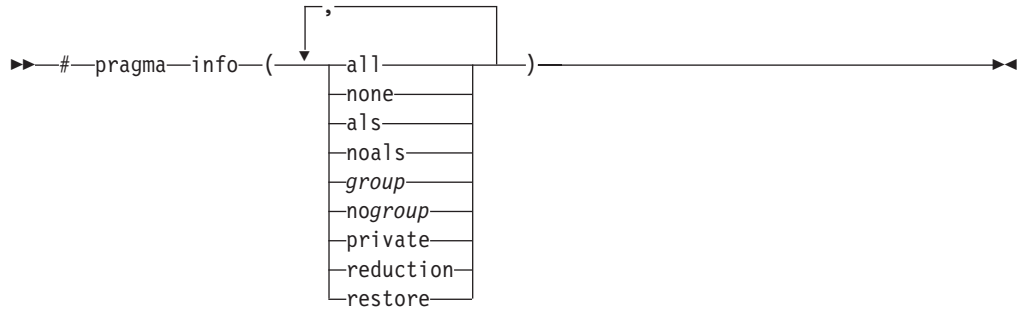
Produces or suppresses groups of informational messages.

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─┘ │
```

Pragma syntax

```
                              ,
                      ┌───◄──────────────┐
►►──#──pragma──info──(──┬──all────────┬──)──────────────────────────►◄
                        ├──none───────┤
                        ├──als────────┤
                        ├──noals──────┤
                        ├──group──────┤
                        ├──nogroup────┤
                        ├──private────┤
                        ├──reduction──┤
                        └──restore────┘
```

Defaults

- ► **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.

private
> Lists shared variables made private to a parallel loop.

reduction
> Lists all variables that are recognized as reduction variables inside a parallel loop.

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.

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**.

Specifying **-qnoinfo** is equivalent to **-qinfo=noall**.

-qalias=ansi must be set before reporting of aliasing-rule violations (**-qinfo=als**) 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()
{
  .
  .
  .

}
```

Related information
- "-qflag" on page 113

-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 **inlglue** is in effect, the optimizer inlines glue code for better performance. When **noinlglue** is in effect, inlining of glue code is prevented.

Syntax

```
►►── -q──┬─noinlglue─┬───────────────────────────────────────────────►◄
         └─inlglue───┘
```

Defaults
- -qnoinlglue
- **-qinlglue** when **-qtune=pwr4** or higher, **-qtune=auto**, or **-qtune=balanced** is in effect (that is, **-qtune=pwr4 | pwr5 | pwr6 | ppc970**, or **-qtune=auto | balanced** is in effect on a machine with the appropriate POWER4, or later processor).

Usage

If you use the **-qtune** option with any of the suboptions that imply **-qinlglue** and you want to disable inlining of glue code, make sure to specify **-qnoinlglue** as well.

Inlining glue code can cause the code size to grow. **-qcompact** overrides the **-qinlglue** setting regardless of other options specified; if you want **-qinlglue** to be enabled, do not specify **-qcompact**.

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 87
- "-qprocimported, -qproclocal, -qprocunknown" on page 225
- "-qtune" on page 283

-qinline

See "-Q, -qinline " on page 228.

-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──┬───────────────────────────────────────────┬───┘
                 │        ┌─noclonearch──────────────┐        │
                 └─=─┬─┬──┴──────────────────────────┴──────┐─┘
                     │ │                        ┌─,─┐        │
                     │ └─clonearch─=─────┴─arch─┴──┘         │
                     │   ┌─nocloneproc─┐   ┌─,─┐             │
                     ├───┴─cloneproc───┴─=─┴─function_name─┘ │
                     │              ┌─,─┐                     │
                     ├─exits─=──────┴─function_name─┘         │
                     ├─inline─┬──────────────────────────────┤
                     │        └─=─┬─noauto───────────────┐    │
                     │            │    ┌─:─┐             │    │
                     │            │    │  ┌─auto──────┐  │    │
                     │            ├────┴──┼─limit──────┼─=─size─┤
                     │            │       └─threshold─┘        │
                     │            │    ┌─,─┐                   │
                     │            └────┴─function_name─┘       │
                     ├─noinline───────────────────────────────┤
                     │        ┌─,─┐                            │
                     │        └─=──┴─function_name─┘           │
                     │                  ┌─,─┐                  │
                     ├─infrequentlabel─=┴─label_name─┘         │
                     │          ┌─1─┐                          │
                     ├─level─=──┼─0─┤                          │
                     │          └─2─┘                          │
                     ├─list─┬──────────────────────┐──────────┤
                     │      └─=─┬─file_name─┐                  │
                     │          ├─long──────┤                  │
                     │          └─short─────┘                  │
                     │             ┌─,─┐                       │
                     ├─lowfreq─=───┴─function_name─┘           │
                     │  ┌─malloc16───┐                         │
                     ├──┴─nomalloc16─┘                         │
                     │            ┌─unknown──┐                 │
                     ├─missing─=──┼─safe─────┤                 │
                     │            ├─isolated─┤                 │
                     │            └─pure─────┘                 │
                     │               ┌─medium─┐                │
                     ├─partition──=──┼─small──┤                │
                     │               └─large──┘                │
                     │  ┌─threads─┬───────────────┐            │
                     ├──┤         │  ┌─auto───┐    │           │
                     │  │         └─=┼─number─┤    │           │
                     │  │            └─noauto─┘    │           │
                     │  └─nothreads────────────────┘           │
                     │                    ┌─,─┐                │
                     ├─┬─isolated─┬──=─────┴─function_name─┘    │
                     │ ├─pure─────┤                            │
                     │ ├─safe─────┤                            │
                     │ └─unknown──┘                            │
                     └─file_name──────────────────────────────┘
```

Defaults

- **-qnoipa**
- **-qipa=inline=auto:level=1:missing=unknown:partition=medium:threads=auto**
  when **-O4** is in effect
- **-qipa=inline=auto:level=2:missing=unknown:partition=medium:threads=auto**
  when **-O5** is in effect

- **-qipa=inline=auto:level=0:missing=unknown:partition=medium:threads=auto**
  when **-qpdf1** or **-qpdf2** is in effect

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
> Specifies the architectures for which multiple versions of the same instruction set are produced.
>
> If **clonearch** is in effect, during the IPA link phase, the compiler generates a generic version of the instruction set based on the **-qarch** setting in effect, and if appropriate, *clones* specialized versions of the instruction set for the architectures you specify in the **clonearch** suboption. The compiler inserts code into your application to check for the processor architecture at run time. When run, the application's version of the instruction set that is best optimized for the runtime environment is selected. Use this suboption if you require optimal performance on multiple differing machines running the same copy of your application.
>
> This suboption is disabled if **-qcompact** is in effect.

arch
> Is a comma-separated list of architectures. The supported values are:
> - pwr4
> - pwr5
> - ppc970
> - pwr6
>
> If you specify no value, an invalid value or a value equal to the **-qarch** setting, no function versioning will be performed for this option. The following table lists allowed **clonearch** for difference architectures.

Table 24. Compatible architecture and clonearch settings

-qarch setting	Allowed clonearch value
ppc, pwr3, ppc64, ppcgr, ppc64gr, ppc64grsq	pwr4, pwr5, ppc970, pwr6
pwr4	pwr5, ppc970, pwr6
ppc64v	ppc970, pwr6
pwr5	pwr6

Table 24. Compatible architecture and clonearch settings (continued)

-qarch setting	Allowed clonearch value
ppc970	pwr6
pwr6	None

To ensure compatibility across multiple platforms, the **-qarch** option must be set to a value that is a subset of the architectures specified by **-qipa=clonearch**. In the case that suboptions are specified for **-qipa=clonearch** and **-qarch** that do not match the target architecture, the compiler will generate instructions based on the suboption that most closely matches the system on which the application is currently running.

You can also use the **-qreport** option in conjunction with **-qipa=clonearch** to produce a report showing how functions were cloned; see "-qreport" on page 233 for details.

cloneproc | nocloneproc
When **-qipa=clonearch** is in effect, **cloneproc** specifies that only the named functions are to be cloned; **nocloneproc** specifies functions that should not be cloned. Note that by default the compiler avoids cloning functions that are specified as low frequency functions with the **lowfreq** suboption.

function_name
For all suboptions, the name of a function, or a comma-separated list of functions.

▶ `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. You can also use can the **c++filt** utility provided by the compiler for a side-by-side listing of source names and mangled names; see "Demangling compiled C++ names" in the *XL C/C++ Optimization and Programming Guide* for details. (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.)

Regular expression syntax can be used to specify names for all suboptions that take function names as suboptions. Syntax rules for specifying regular expressions are described below:

Expression	Description
string	Matches any of the characters specified in *string*. For example, `test` will match `testimony`, `latest`, and `intestine`.
^*string*	Matches the pattern specified by *string* only if it occurs at the beginning of a line.
string$	Matches the pattern specified by *string* only if it occurs at the end of a line.
str.*ing*	The period (.) matches any single character. For example, `t.st` will match `test`, `tast`, `tZst`, and `t1st`.
string*special_char*	The backslash (\ ) can be used to escape special characters. For example, assume that you want to find lines ending with a period. Simply specifying the expression `.$` would show all lines that had at least one character of any kind in it. Specifying `\.$` escapes the period (.), and treats it as an ordinary character for matching purposes.

Expression	Description
[*string*]	Matches any of the characters specified in *string*. For example, `t[a-g123]st` matches `tast` and `test`, but not `t-st` or `tAst`.
[^*string*]	Does not match any of the characters specified in *string*. For example, `t[^a-zA-Z]st` matches `t1st`, `t-st`, and `t,st` but not `test` or `tYst`.
string*	Matches zero or more occurrences of the pattern specified by *string*. For example, `te*st` will match `tst`, `test`, and `teeeeeest`.
string+	Matches one or more occurrences of the pattern specified by *string*. For example, `t(es)+t` matches `test`, `tesest`, but not `tt`.
string?	Matches zero or one occurrences of the pattern specified by **string**. For example, `te?st` matches either `tst` or `test`.
string{*m*,*n*}	Matches between *m* and *n* occurrence(s) of the pattern specified by *string*. For example, `a{2}` matches `aa`, and `b{1,4}` matches `b`, `bb`, `bbb`, and `bbbb`.
string1 | *string2*	Matches the pattern specified by either *string1* or *string2*. For example, `s | o` matches both characters `s` and `o`.

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.

inline

Enables function inlining by the high-level optimizer. Valid suboptions are any of the following:

auto | **noauto**

Enables or disables automatic function inlining by the high-level optimizer. When **-qipa=inline=auto** is in effect, the compiler considers all functions that are under the maximum size limit (see below) for inlining. When **-qipa=inline=noauto** is in effect, only functions listed in the *function_name* suboption are considered for inlining.

limit

When **-qipa=inline=auto** is in effect, specifies a limit on the size of a calling function after inlining.

threshold

When **-qipa=inline=auto** is in effect, specifies a limit on the size of a called function for it to be considered for inlining.

size

A nonnegative integer representing the relative size of function before and after inlining. The *size* is an arbitrary value representing a combination of factors, including the estimated size of the called function, the number of calls to the function, and so on. If you do not specify a *size*, the default is

1024 for the **threshold** suboption and 8192 for the **limit** suboption. Larger values for this number allow the compiler to inline larger functions, more function calls, or both.

Specifying **-qipa=inline** with no suboptions is equivalent to **-qipa=inline=auto**.

Note: By default, the compiler will try to inline all functions, not just those that you specified with the *function_name* suboption. If you want to turn on inlining for only certain functions, specify **inline=noauto** after you specify **inline=***function_name*. (You must specify the suboptions in this order.) For example, to turn off inlining for all functions other than for sub1, specify **-qipa=inline=sub1:inline=noauto**.

noinline
When specified with no suboption, disables automatic function inlining by the high-level optimizer (equivalent to **-qipa=inline=noauto**). (Inlining may still be performed by the compiler front end or by the low-level optimizer; see "-Q, -qinline " on page 228 for details.) When used with the *function_name* suboption, specifies functions that are not to be considered for automatic inlining by the high-level optimizer.

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.

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.

malloc16 | nomalloc16
Informs the compiler that the dynamic memory allocation routines will return 16-byte aligned memory addresses. The compiler can then optimize the code based on that assertion.

In 64-bit mode, AIX always returns 16-byte aligned addresses and therefore by default **-qipa=malloc16** is in effect. You can use **-qipa=nomalloc16** to override the default setting.

Note: You must make sure that the executables generated with **-qipa=malloc16** run in an environment in which dynamic memory allocations return 16-byte aligned addresses, otherwise, wrong results can be generated. For example, in 32-bit mode, addresses are not 16-byte aligned. In this case, you must set the MALLOCALIGN=16 runtime environment variable.

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}
clonearch=arch,{arch}
cloneproc=name,{name}

nocloneproc=name,{name}missing = attribute{, attribute}
exits = name{, name}
lowfreq = name{, name}
inline
inline [ = auto | = noauto ]
inline = name{, name} [ from name{, name}]
inline-threshold = unsigned_int
inline-limit = unsigned_int
list [ = file-name | short | long ]
noinline
noinline = name{, name} [ from name{, name}]
level = 0 | 1 | 2
partition = small | medium | large
```

where *attribute* is one of:
- clonearch
- cloneproc
- nocloneproc
- exits
- lowfreq
- unknown

- safe
- isolated
- pure

Specifying **-qipa** with no suboptions on the link step is equivalent to **-qipa=inline=auto:level=1:missing=unknown:partition=medium:threads=auto**.

Note: 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 215 for details.

Usage

Specifying **-qipa** automatically sets the optimization level to **-O2**. For additional performance benefits, you can also specify the **-Q** 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**, **dump**, 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

- "-Q, -qinline " on page 228
- "-qisolated_call"
- "#pragma execution_frequency" on page 322
- "-qpdf1, -qpdf2" on page 215
- "-S" on page 241
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*

-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

-qkeepinlines (C++ only)

Category

Object code control

Pragma equivalent

None.

Purpose

Keeps or discards definitions for unreferenced extern inline functions.

When **-qnokeepinlines** is in effect, definitions of unreferenced external inline functions are discarded. When **-qkeepinlines** is in effect, definitions of unreferenced external inline functions are kept.

Syntax

```
                ┌─nokeepinlines─┐
►►── -q─────────┴─keepinlines───┴──────────────────────────────────────────►◄
```

Defaults

-qnokeepinlines

Usage

-qnokeepinlines reduces the size of the object files. **-qkeepinlines** provides the same behavior as VisualAge C++ compilers previous to the v5.0.2.1 update level, allowing compatibility with shared libraries and object files built with the earlier releases of the compiler.

Predefined macros

None.

Related information
* "-qstaticinline (C++ only)" on page 256

-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 204

-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 71

-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** or **-Z**

option. See "-B" on page 76, "-brtl" on page 78, and "-b" on page 75 for information on specifying the types of libraries that are searched (for static or dynamic linking).

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"
- "-b" on page 75
- "-brtl" on page 78
- "-Z" on page 307
- "Specifying compiler options in a configuration file" on page 7

-L

Category

Linking

Pragma equivalent

None.

Purpose

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

When you link shared libraries into an executable, specifying the paths to the libraries with the **-L** option during the link also embeds the path information in the executable, so the shared libraries can be correctly located at run time. If you do not specify any paths with **-L** during this link and you additionally prevent the compiler from automatically passing **-L** arguments to the linker by using the **-bnolibpath** linker option, only paths that are specified by the LIBPATH environment variable are embedded in the executable file.

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.

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 163

-qlanglvl

Category

Language element control

Pragma equivalent

▶ **C** ◄ #pragma options langlvl, #pragma langlvl

▶ **C++** ◄ See also #pragma operator_new

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

```
                              :
                     ┌─────────────────┐
              ┌─▼──────────────────────┐  ┌──────┐
►►── -q─langlvl─=──▼──┬─extc99──┐──────┬──┬─ucs───┬────────────────►◄
                     ├─classic─┤      └──┴─noucs──┘
                     ├─extc89──┤
                     ├─extended┤
                     ├─saa─────┤
                     ├─saal2───┤
                     ├─stdc89──┤
                     └─stdc99──┘
```
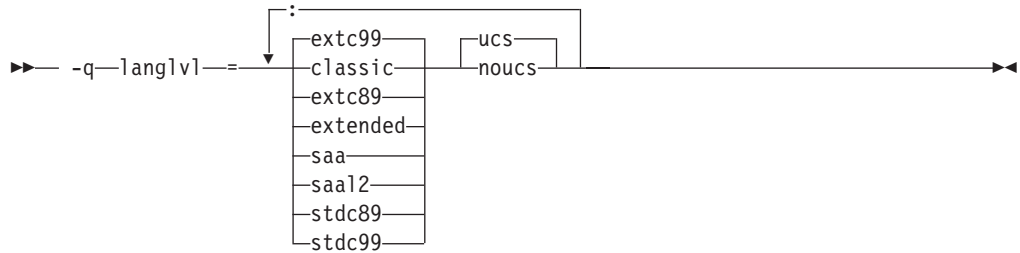
#pragma langlvl syntax — C only

```
                              ┌─extc99───┐
►►──#──pragma──langlvl──(──┬─classic──┬──)───────────────────────────────►◄
                           ├─extc89───┤
                           ├─extended─┤
                           ├─saa──────┤
                           ├─saal2────┤
                           ├─stdc89───┤
                           └─stdc99───┘
```

-qlanglvl syntax — C++

```
                                          :
                        ┌─extended──┐  ┌──────────┐
►►── -q──langlvl──=──┬─compat366─┬──▼─feature_suboption─┬──────────────►◄
                        ├─extended0x┤  └──────────────────┘
                        └─strict98──┘
```
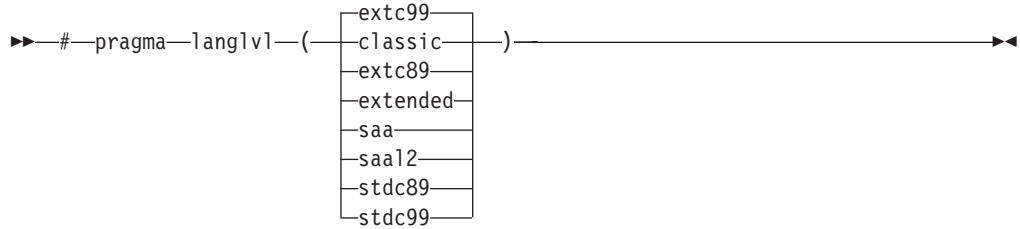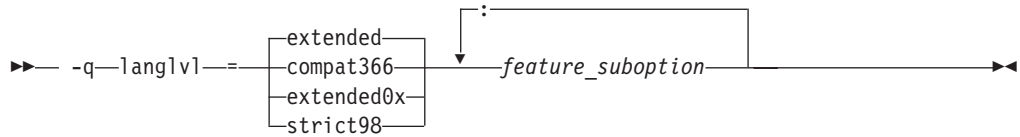
Defaults

- ▶ 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++

  -qlanglvl=extended:anonstruct:anonunion:ansifor:ansisinit:c99__func__:noc99complex:noc99comple

Parameters

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. This language level is not supported by the AIX V5.1
and higher system header files, such as math.h. If you use the AIX V5.1 or
higher system header files, consider compiling your program to the **stdc89** or
extended language levels.

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.

> **Note:** Not all operating system releases support the header files and runtime library required by C99.

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.

The following are the **-qlanglvl** parameters for C++ language programs:

compat366
Compilation conforms to some, but not all, IBM C++ Compiler V3.6 features.

extended
Compilation is based on the ISO C++ standard, with some differences to accommodate extended language features.

extended0x
A new group option has been created to compile code using all the features of the new version of the C++ standard (C++0x) that are implemented in IBM XL C/C++ for AIX, V10.1. Specifically, in this release we have introduced new integer promotion rules for arithmetic conversions with `long long` data types and we have enhanced the C++ preprocessor to support C99 features.

For more details about the new integer promotion rules, see "Integer promotion rules for C99 long long under C++" in the *XL C/C++ Language Reference*.

For more details about preprocessor changes, see "Preprocessor changes" in the *Getting Started with XL C/C++*.

Note: C++0x is a new version of the standard for the C++ programming language. This standard is temporary and has not been officially adopted in its entirety. Be aware that any future level of support for this standard is likely to change. Also see "Language levels and language extensions" in the *XL C/C++ Language Reference*

strict98
Compilation conforms strictly to the ISO C++ standard.

feature_suboption
Can be any of the following:

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.

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.

c99complexheader | noc99complexheader

Enables or disables use of the C99 complex.h header file.

c99compoundliteral | noc99compoundliteral
Enables or disables support for C99 compound literals.

c99hexfloat | noc99hexfloat
Enables or disables support for C99-style hexadecimal floating constants.

c99vla | noc99vla
Enables or disables support for C99-type variable length arrays.

compatzea | nocompatzea
Controls whether zero extent arrays have an underlying dimension of 1 or 0. When the default, **-qlanglvl=nocompatzea**, is in effect, zero extent arrays have a dimension of 0. Use **-qlanglvl=compatzea** to specify that zero extent arrays should have a dimension of 1, for compatibility with code compiled with VisualAge C++ V6.0 and predecessor products. Specifying **-qlanglvl=compatzea** has effect only if **-qlanglvl=zeroextarray** is also in effect.

dependentbaselookup | nodependentbaselookup
Controls whether the name lookup rules for a template base class of dependent type defined in the TC1 of the C++ Standard apply. When the default, **-qlanglvl=dependentbaselookup**, is in effect, a member of a base class that is a dependent type hides a name declared within a template or any name from within the enclosing scope of the template. This setting is compatible with the behavior of previous XL C++ compilers. Specify **-qlanglvl=nodependentbaselookup** for compliance with TC1.

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*.

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

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;
}
```

newexcp | nonewexcp
Controls whether the `new` operator throws an exception when the requested memory fails. When the default, **-qlanglvl=nonewexcp**, is in effect, the null pointer 0 is returned. When **-qlanglvl=newexcp** is in effect, the standard exception `std::bad_alloc` is thrown. For compatibility with earlier versions of VisualAge C++ and predecessor products, specify **-qlanglvl=nonewexcp**. For conformance to the C++ standard, which fully supports `new` exceptions, specify **-qlanglvl=newexcp**.

This suboption does not apply to the `nothrow` versions of the `new` operator, `new` operators with empty `throw` specifications, class-specific `new` operators, and `new` operators with placement arguments.

Note: You can also use the equivalent **#pragma operator_new** directive to specify this suboption for selected portions of code. See "#pragma operator_new (C++ only)" on page 339 for details.

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;
> ```

oldmath | nooldmath
> Controls the versions of math function declarations in math.h that are included when you specify math.h as an included or primary source file.
>
> Specify **-qlanglvl=nooldmath** for strict compliance with the C++ standard. Specify **-qlanglvl=oldmath** for compatibility with earlier versions of VisualAge C++ and predecessor products.

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 | <u>nooldtmplalign</u>

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.

<u>oldtmplspec</u> | 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 | <u>noredefmac</u>

Controls whether a macro can be redefined without a prior #undef or undefine() statement.

<u>trailenum</u> | 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. For details of this feature, see "Function-like macros" in the *XL C/C++ Language Reference*.

zeroextarray | nozeroextentarray
Controls whether zero-extent arrays are allowed as the last non-static data member in a class definition. When the default, **-qlanglvl=zeroextentarray**, 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 385 for a list of macros that are predefined by **-qlanglvl** suboptions.

Related information
- "-qsuppress" on page 265
- "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*

-qlargepage
Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Takes advantage of large pages provided on POWER4 and higher systems, for applications designed to execute in a large page memory environment.

When **-qlargepage** is in effect to compile a program designed for a large page environment, an increase in performance can occur.

Syntax

```
         ┌─nolargepage─┐
►►── -q───┴─largepage──┴──────────────────────────────────────────►◄
```

Defaults

-qnolargepage

Usage

Note that this option is only useful in the following conditions:
- Large pages must be available and configured on the system.
- You must compile with an option that enables loop optimization, such as **-O3** or **-qhot**.
- You must link with the **-blpdata** option.

See your AIX operating system documentation for more information on using large page support.

Predefined macros

None.

Examples

To compile `myprogram.c` to use large page heaps, enter:

```
xlc myprogram.c -qlargepage -blpdata
```

-qldbl128, -qlongdouble

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

```
►►── -q─┬─nolongdouble─┬──────────────────────────────────────────────►◄
        ├─noldbl128────┤
        ├─ldbl128──────┤
        └─longdouble───┘
```

Defaults

-qnoldbl128

Usage

Separate libraries are provided that support 128-bit `long double` types. These libraries will be automatically linked if you use any of the invocation commands with the **128** suffix (**xlc++128**, **xlc128**, **cc128**, **xlc++128_r**, **xlc128_r,** or **cc128_r**). You can also manually link to the 128-bit versions of the libraries using the **-l***key* option, as shown in the following table:

Default (64-bit) long double		128-bit long double	
Library	**Form of the -l***key* **option**	**Library**	**Form of the -l***key* **option**
libC.a	-lC	libC128.a	-lC128
libC_r.a	-lC_r	libC128_r.a	-lC128_r

Linking without the 128-bit versions of the libraries when your program uses 128-bit long doubles (for example, if you specify **-qldbl128** alone) may produce unpredictable results.

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 is defined to 1 when **-qldbl128** is in effect; otherwise, it is 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 -lC128
```

Related information
- "-l" on page 163

-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

```
►►─ -q─┬─lib──┬─────────────────────────────────►◄
        └─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/aix51/, lib/aix52/, and lib/aix53/ 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 89

-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

```
           ┌─nolibansi─┐
►►── -q────┴─libansi───┴──────────────────────────────────────────►◄
```

Defaults

-qnolibansi

Predefined macros

▶ C++ ◀ __LIBANSI__ is defined to 1 when **libansi** is in effect; otherwise, it is not defined.

-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

```
           ┌─nolinedebug─┐
►►── -q────┴─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 126
- "-O, -qoptimize" on page 204

-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 20.

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---'
```
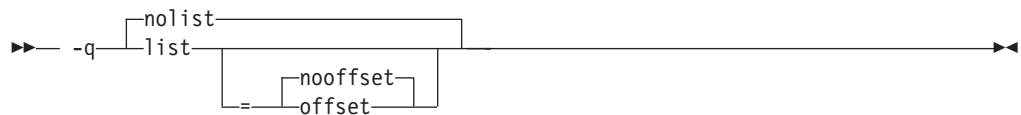
Defaults

-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 184
- "-qprint" on page 223
- "-qsource" on page 251

-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 20.

Syntax

```
>>-- -q--┬-nolistopt-┬-----------------------------------------><
         └-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 182
- "-qprint" on page 223
- "-qsource" on page 251

-qlonglit

Category

Floating-point and integer control

Pragma equivalent

None.

Purpose

In 64-bit mode, promotes literals with implicit type of int to long.

Syntax

```
►►── -q──┬─nolonglit─┬──────────────────────────────────────────►◄
         └─longlit───┘
```

Defaults

-qnolonglit

Usage

The following table shows the default implicit types for constants and the implicit types when **-qlonglit** is in effect.

Suffix	Decimal literals		Hexadecimal or octal literals	
	Default implicit type	Implicit type with -qlonglit in effect	Default implicit type	Implicit type with -qlonglit in effect
unsuffixed	int long int	long int	int unsigned int long int unsigned long int	long int unsigned long int
u or U	unsigned int unsigned long int	unsigned long int	unsigned int unsigned long int	unsigned long int
l or L	long int	long int	long int unsigned long int	long int unsigned long int
Both u or U, and l or L	unsigned long int	unsigned long int	unsigned long int	unsigned long int
ll or LL	long long int	long long int	long long int unsigned long long int	long long int unsigned long long int
Both u or U, and ll or LL	unsigned long long int	unsigned long long int	unsigned long long int	unsigned long long int

Predefined macros

None.

-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 **cc** invocation command or the **-qlanglvl=extended | extc89** option; **-qnolonglong** for the **c89** invocation command or **-qlanglvl=stdc89** option.
- **C++** **-qlonglong**; **-qnolonglong** when **-qlanglvl=strict98** is in effect.

Usage

C This option only has an effect with the **cc** or **c89** invocation commands, or when the **-qlanglvl** option is set to **extended | stdc89 | extc89**. It is not valid for the **xlc** invocation command or when the language level **stdc99 | extc99** is in effect, as the `long long` support provided by this option is incompatible with the semantics of the `long long` types mandated by the C99 standard.

C++ The functionality provided by this option is available with the **-qlanglvl=extended0x** option, as well. The only difference is that new unsigned integer promotion rules apply with that option. For more details about those rules see "Integer promotion rules for C99 long long under C++" in the *XL C/C++ Language Reference*.

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
```

AIX v4.2 and later provides support for files greater than 2 gigabytes in size so you can store large quantities of data in a single file. To allow large file manipulation in your application, compile with the -D_LARGE_FILES and **-qlonglong** compiler options. For example:
```
xlc myprogram.c -D_LARGE_FILES -qlonglong
```

Related information

- "Integral types" in the *XL C/C++ Language Reference*

-ma (C only)

See "-qalloca, -ma (C only)" on page 65.

-qmacpstr

Category

Language element control

Pragma equivalent

#pragma options [no]macpstr

Purpose

Converts Pascal string literals (prefixed by the \p escape sequence) into null-terminated strings in which the first byte contains the length of the string.

For example, when the **-qmacpstr** option is in effect, the compiler converts:
```
"\pABC"
```

to:
```
'\03' , 'A' , 'B' , 'C' , '\0'
```

Syntax

```
                  ┌─nomacpstr─┐
►►─── -q───────────┴─macpstr───┴──────────────────────────────────────►◄
```

Defaults

-qnomacpstr

Usage

A Pascal string literal always contains the characters "\p. The characters \p in the middle of a string do not form a Pascal string literal; the characters must be *immediately preceded* by the " (double quote) character.

Entering the characters:
```
'\p' , 'A' , 'B' , 'C' , '\0'
```

into a character array does not form a Pascal string literal.

The compiler ignores the **-qmacpstr** option when the **-qmbcs** or **-qdbcs** option is active because Pascal-string-literal processing is only valid for one-byte characters.

The **#pragma options** keyword **macpstr** is only valid at the top of a source file before any C or C++ source statements. If you attempt to use it in the middle of a source file, it is ignored and the compiler issues an error message.

The following describes how Pascal string literals are processed.

- Because there is no Pascal-string-literal processing of wide strings, using the escape sequence \p in a wide string literal with the **-qmacpstr** option, generates a warning message and the escape sequence is ignored.
- Concatenating a Pascal string literal to a normal string gives a non-Pascal string. For example, concatenating the strings:

```
"ABC" "\pDEF"
```

gives:

```
"ABCpDEF"
```

- Concatenating two Pascal string literals, for example, `strcat`, does not result in a Pascal string literal. However, as described above, two adjacent Pascal string literals can be concatenated to form one Pascal string literal in which the first byte is the length of the new string literal. For example, concatenating the strings:

```
"\p ABC" "\p DEF"
```

or

```
"\p ABC" "DEF"
```

results in:

```
"\06ABCDEF"
```

- A Pascal string literal cannot be concatenated with a wide string literal.
- The compiler truncates a Pascal string literal that is longer than 255 bytes (excluding the length byte and the terminating NULL) to 255 characters.
- The Pascal string literal is *not* a basic type different from other C or C++ string literals. After the processing of the Pascal string literal is complete, the resulting string is treated the same as all other strings. If the program passes a C string to a function that expects a Pascal string, or vice versa, the behavior is undefined.
- Modifying any byte of the Pascal string literal after the processing has been completed does not alter the original length value in the first byte. For example, in the string "\06ABCDEF", substituting a null character for one of the existing characters in the middle of the string does not change the value of the first byte of the string, which contains the length of the string.
- No errors or warnings are issued when the bytes of the processed Pascal string literal are modified.

Predefined macros

None.

Examples

To compile mypascal.c and convert string literals into Pascal-style strings, enter:

```
xlc mypascal.c -qmacpstr
```

Related information
- "-qmbcs, -qdbcs" on page 193

-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 .u 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 .u 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 13. If the include file is not found, it is not added to the .u file.

Files with no include statements produce output files containing one line that lists only the input file name.

Predefined macros

None.

Examples

To compile `mysource.c` and create an output file named `mysource.u`, enter:

```
xlc -c -qmakedep mysource.c
```

To compile `foo_src.c` and create an output file named `mysource.u`, enter:

```
xlc -c -qmakedep foo_src.c -MF mysource.u
```

To compile `foo_src.c` and create an output file named `mysource.u` in the `deps/` directory, enter:

```
xlc -c -qmakedep foo_src.c -MF deps/mysource.u
```

To compile `foo_src.c` and create an object file named `foo_obj.o` and an output file named `foo_obj.u`, 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.u`, enter:

```
xlc -c -qmakedep foo_src.c -o foo_obj.o -MF mysource.u
```

To compile `foo_src1.c` and `foo_src2.c` to create two output files, named `foo_src1.u` and `foo_src2.u`, respectively, in the `c:/tmp/` directory, enter:

```
xlc -c -qmakedep foo_src1.c foo_src2.c -MF c:/tmp/
```

Related information
- "-MF" on page 194
- "-o" on page 203
- "Directory search sequence for include files" on page 13

-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

```
                 ┌─nomaxerr─────────────────────────────┐
►►── -q──────────┼─maxerr──=──number─────────────────────┼──────────────────────────►◄
                                    │   ┌─s─┐ │
                                    └─:─┼─i─┼─┘
                                        ├─w─┤
                                        └─e─┘
```

-qmaxerr syntax — C++

```
                 ┌─nomaxerr─────────────────────────────┐
►►── -q──────────┼─maxerr──=──number─────────────────────┼──────────────────────────►◄
                                   │   ┌─s─┐ │
                                   └─:─┼─i─┼─┘
                                       └─w─┘
```
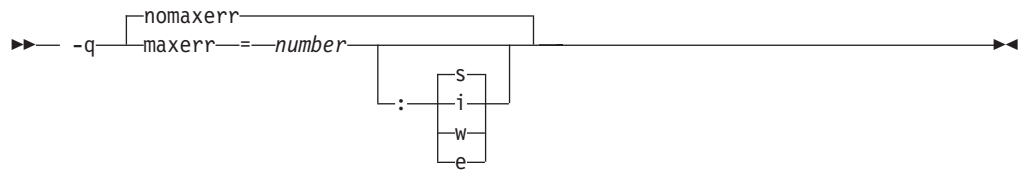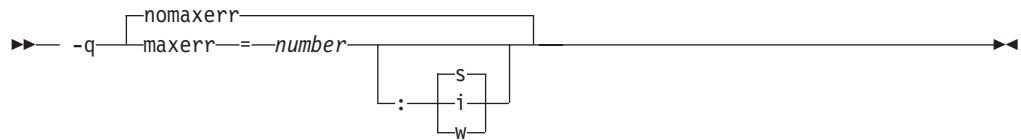
Defaults

-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 113
- "-qhalt" on page 129
- "Message severity levels and compiler response" on page 18

-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 -omultibyte_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="kpsmultibyte_chardcs"
  -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 92

-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 188 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 188
- "-o" on page 203
- "Directory search sequence for include files" on page 13

-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.

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

```
►►── -q──┬─nominimaltoc─┬──────────────────────────────────────────────────►◄
         └─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.

You should use this option, together with the related options described below, instead of calling the linker directly (or using the **makeC++SharedLib** utility, in C++) 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 — C

```
►►── -q──mkshrobj──────────────────────────────────────────────────────────►◄
```

-qmkshrobj syntax — C++

```
►►── -q─mkshrobj─────────────────────────────────────────────────►◄
              └─=─priority─┘
```

Defaults

By default, the output object is linked with the runtime libraries and startup routines to create an executable file.

Parameters

▶ **C++** *priority*
> Specifies the priority level for the initialization order of static C++ objects declared in the shared object (relative to static objects declared in other shared objects). The *priority* may be any number from -214 782 624 (highest priority — initialized first) to 214 783 647 (lowest priority — initialized last). Numbers from -214 783 648 to -214 782 623 are reserved for system use. If no priority is specified a default priority of 0 is used. The priority has no effect if you link shared objects written in C, if you link with the C compiler (**xlc**), or if the shared object has no static initialization.

Usage

The compiler will automatically export all global symbols from the shared object unless you explicitly specify which symbols to export with the **-bE:**, **-bexport:** or **-bnoexpall** options, or if you use the **-qnoweakexp** option to prevent weak symbols from being exported.

Specifying **-qmkshrobj** implies **-qpic**.

You can also use the following related options with the **-qmkshrobj**:

-o *shared_file*
> The name of the file that will hold the shared file information. The default is shr.o.

-qexpfile=*filename*
> Saves all exported symbols in *filename*.

-e *name*
> Sets the entry name for the shared executable to *name*. The default is **-enoentry**.

-q[no]weakexp
> Specifies whether symbols marked as weak (with the **#pragma weak** directive) are to be included in the export list. If you do not explicitly set this option, the default is **-qweakexp** (global weak symbols are exported).

For detailed information on using **-qmkshrobj** to create shared libraries, as well as examples of using **-qmkshrobj** with priority values, 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, type:

```
xlc -qmkshrobj -o big_lib.so lib_a.o lib_b.o lib_c.o
```

Related information

- "-o" on page 203
- "-e" on page 101
- "-qpriority (C++ only)" on page 224
- "-qpic" on page 221
- "-b" on page 75
- "-qexpfile" on page 108
- "-qweakexp" on page 302

-qnamemangling (C++ only)
Category

Portability and migration

Pragma equivalent

#pragma namemangling

Purpose

Chooses the name mangling scheme for external symbol names generated from
C++ source code.

The option and pragma are provided to ensure binary compatibility with link
modules created with previous versions of the compiler. If you do not need to
ensure backwards compatibility, it is recommended that you do not change the
default setting of this option.

Syntax

Option syntax

```
►►── -q─namemangling─=──┬─ansi───┬───────────────────────►◄
                        ├─v10────┤
                        ├─v9─────┤
                        ├─v8─────┤
                        ├─v7─────┤
                        ├─v6─────┤
                        ├─v5─────┤
                        ├─v4─────┤
                        ├─v3─────┤
                        └─compat─┘
```

Pragma syntax

```
►►──#──pragma──namemangling──(──┬─────┬──ansi───┬──────┬──────────────────┬───────)──────────────►◄
                                │     ├──v10──────┤      └─,──num_chars──┘
                                │     ├──v9───────┤
                                │     ├──v8───────┤
                                │     ├──v7───────┤
                                │     ├──v6───────┤
                                │     ├──v5───────┤
                                │     ├──v4───────┤
                                │     ├──v3───────┤
                                │     └──compat───┘
                                └──pop─────────────
```
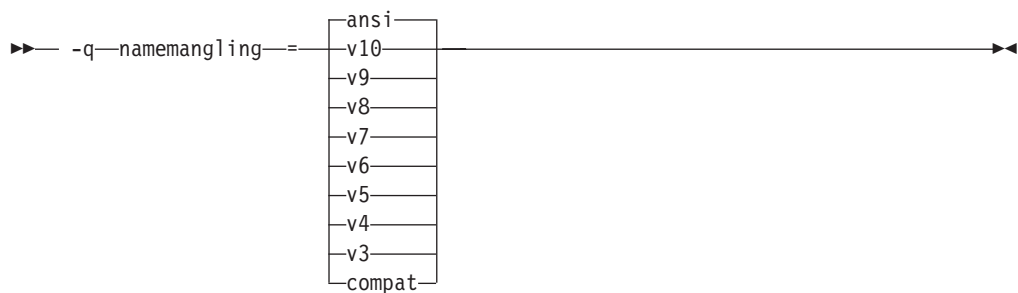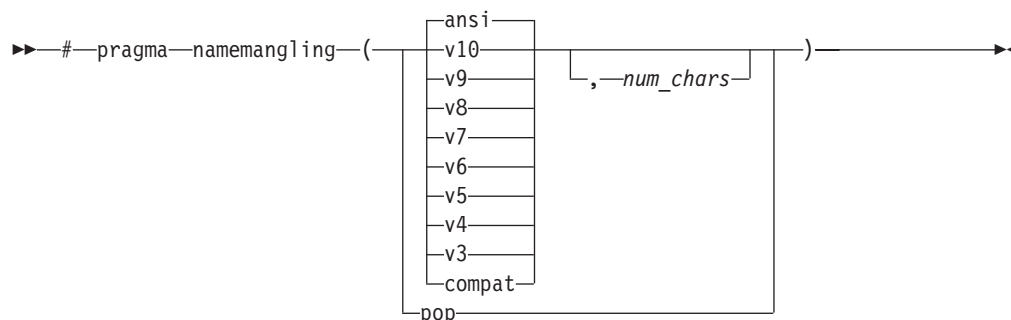
Defaults

-qnamemangling(ansi, 64000)

Parameters

ansi
> The name mangling scheme fully supports the most recent language features of Standard C++, including function template overloading.

v3 | compat
> The name mangling scheme is compatible with VisualAge C++ V3.0 in 32-bit mode only.

v4 The name mangling scheme is compatible with the GA release of VisualAge C++ V4.0. Prior to this release, a function and a function template specialization with the same name and parameter list were considered to have the same signature, and the following test case would fail to compile:

```
int f(int) {
    return 42;
}

template < class T > int f(T) {
    return 43;
}

int main() {
    f < int > (3); // instantiate int f < int > (int)
    return f(4);
}
```

From V4.0 on, the compiler treats a function and a function template specialization with the same name and parameter list as distinct functions. The following examples illustrate this behavior:

Source name	Mangled name prior to v4	Mangled name in v4 and higher
int f (int)	f__Fi	f__Fi
int f <int> (int)	f__Fi	f__Hi_i_i

v5 The name mangling scheme is compatible with the GA release of VisualAge C++ V5.0. Same as the **v4** suboption.

v6 The name mangling scheme is compatible with the GA release of VisualAge C++ V6.0. Prior to this release, top-level cv-qualifiers in function arguments were encoded in mangled names. From V6.0 on, in accordance with the C++ Standard, top-level cv-qualifiers are not considered part of the underlying type

of a function argument, and the cv-qualifiers are not encoded in the mangled
names. The following examples illustrate this behavior:

Source name	Mangled name prior to v6	Mangled name in v6 and higher
void foo (const int)	foo__FCi	foo__Fi
void foo (int* const)	foo__FCPi	foo__FPi

> **Note:** This behavior can also be controlled with the use of the
> **nameManglingRule(fnparmtype)** pragma directive. For more information, see
> "#pragma namemanglingrule (C++ only)" on page 336.

v7 The name mangling scheme is compatible with the GA release of IBM XL
C/C++ V7.0.

Several changes to the mangling scheme went into effect in IBM XL C/C++
V7.0. First of all, prior to V7.0, top-level cv-qualifiers were used to distinguish
between types in repeated parameters in a function's signature. From V7.0 on,
in accordance with the C++ Standard, top-level cv-qualifiers are ignored for
determining the equivalence between function parameters. Parameters that are
only differentiated by the presence of a top-level cv-qualifier are considered to
be equivalent, and are represented in the compressed encoding scheme used
for repeated parameters of the same type. The following examples illustrate
this behavior:

Source name	Mangled name prior to v7	Mangled name in v7 and higher
void foo (int, const int)	foo__FiCi (pre-v6) foo__Fii (v6)	foo__FiT1
void foo (int* const, int* const)	foo__FCPiCCPi (pre-v6) foo__FPiPi (v6)	foo__FPiT1

> **Note:** This behavior can also be controlled with the use of the
> **nameManglingRule(fnparmtype)** pragma directive. For more information, as
> well as details of the compressed mangling scheme, see "#pragma
> namemanglingrule (C++ only)" on page 336.

Secondly, prior to V7.0, non-type integral template arguments were mangled as
32-bit `unsigned` decimal numbers prefixed by SP. Due to ambiguities
introduced by this in mangling 64-bit values, this scheme has been changed to
the following:

```
non-type template argument → SM          #single repeat of a previous parameter
                           → SP number #positive internal argument
                           → SN number #negative internal argument
```

When a non-type integral template argument is positive, the number is
prefixed with SP. When a non-type integral template argument is negative, the
number is prefixed with SN, and the decimal number is written without the
minus sign. There is no limit in the range of decimal numbers which can be
represented. The following examples illustrate this behavior:

Source name	Mangled template name prior to v7	Mangled template name in v7 and higher
template <int n> int foo() { return N; } int main() { return foo<-3>(); }	foo__HxiSP429	foo__HxiSN3x_v

v8 The name mangling scheme is compatible with the GA release of IBM XL C/C++ V8.0.

Several changes to the mangling scheme went into effect in IBM XL C/C++ V8.0. First of all, prior to V8.0, intermediate-level cv-qualifiers were not used to distinguish between types in repeated parameters in a function's signature. From V8.0 on, intermediate-level cv-qualifiers are used for determining the equivalence between function parameters. Parameters that are differentiated by the presence of an intermediate-level cv-qualifier are not considered to be equivalent, and are mangled as separate parameters. The following examples illustrate this behavior:

Source name	Mangled name prior to v8	Mangled name in v8 and higher
void foo (int**, int* const *)	foo__FPPiT1	foo__FPPiPCPi

Note: This behavior can also be controlled with the use of the **nameManglingRule(fnparmscmp)** pragma directive. For more information, as well as details of the compressed mangling scheme, see "#pragma namemanglingrule (C++ only)" on page 336.

Secondly, prior to V8.0, only the underlying type in a typedef definition was used to distinguish between types in repeated parameters in a function's signature. From V8.0 on, the name defined in a typedef declaration in a template parameter is encoded as a separate parameter in the mangled name of a template function that uses the typedef as a parameter. The following examples illustrate this behavior:

Source name	Mangled function name prior to v8	Mangled function name in v8 and higher
template <typename T> struct A { typedef int foo; }; template <typename V> int bar (A <V>, int, typename A<V>::foo) {} A<int> a; int x = bar (a, 1, 10);	bar__Hi_1AXTi_iT2_i	bar__Hi_1AXTi_iQ2_1AXTi_9foo_i

Source name	Mangled function name prior to v8	Mangled function name in v8 and higher
template <typename T> struct A { typedef A foo; }; template <typename Y> int bar (A <int>::foo, const A<Y>) {} A<int> a; int x = bar (10, a);	bar__Hi_1AXTi_T1_i	bar__Hi_Q2_1AXTi_foo1AXTi__i

v9 The name mangling scheme is compatible with the GA release of IBM XL C/C++ V9.0. This suboption is equivalent to **ansi**.

Prior to this release, the name mangling scheme did not different between different pointer-to-member template arguments in template instantiations, and the following test case would fail to compile:

```
struct pair {    int x, y;
pair(int x_, int y_) : x(x_), y(y_) {} };

template <int pair::*PtrToPairMember>

struct foo {    int bar(pair& p) { return p.*PtrToPairMember; }
};

template <int pair::*PtrToPairMember> func(pair& p) { return p.*PtrToPairMember; }

int main() {

pair p(0, 1);
foo<&pair::x> fx;
foo<&pair::y> fy;

if (fx.bar(p) != 0 || fy.bar(p) != 1) { return 1; }

if (func<&pair::x>(p) != 0 || func<&pair::y>(p) != 1) { return 2; }

return 0;
}
```

From V9.0 on, the compiler treats different pointer-to-member template arguments as distinct. The following examples illustrate this behavior:

Source name	Mangled name prior to v9	Mangled name in v9and higher
int foo<&pair::y>::bar(pair &)	bar_3fooXA0_FR4pair	bar_3fooXAM1y_FR4pair
int foo<&pair::x>::bar(pair &)	bar_3fooXA0_FR4pair	bar_3fooXAM1x_FR4pair
int func<&pair::y>(pair &)	func_HxM4pairiA0x_R4pair_i	func_HxM4pairiA0yx_R4pair_i
int func<&pair::x>(pair &)	func_HxM4pairiA0x_R4pair_i	func_HxM4pairiA0xx_R4pair_i

v10
The name mangling scheme is compatible with the GA release of IBM XL C/C++ V10.1. This suboption is equivalent to **ansi**.

num_chars **(pragma only)**
Specifies the maximum number of allowable characters in the mangled names.

If you do not specify this suboption, the default maximum is 64000 characters for all settings except **v3** and **compat**, for which the default maximum is 255 characters.

pop
> 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.

Predefined macros

None.

Related information
- "#pragma namemanglingrule (C++ only)" on page 336

-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 79
- "-E" on page 101
- "-P" on page 213
- "-qsyntaxonly (C only)" on page 267

-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


```
                ┌─noopt─────────────┐
                ├─nooptimize────────┤
     ►►──┬─-q─┬─┬─optimize─┬───┬───────────┬──────────────────────────────────►◄
         │    │ └─opt──────┘   └─=─┬───0─┬─┘
         │    │                    ├───2─┤
         │    │                    ├───3─┤
         │    │                    ├───4─┤
         │    │                    └───5─┘
         ├─-O0────────────────────────────────────┤
         ├─-O─────────────────────────────────────┤
         ├─-02────────────────────────────────────┤
         ├─-03────────────────────────────────────┤
         ├─-04────────────────────────────────────┤
         └─-05────────────────────────────────────┘
```
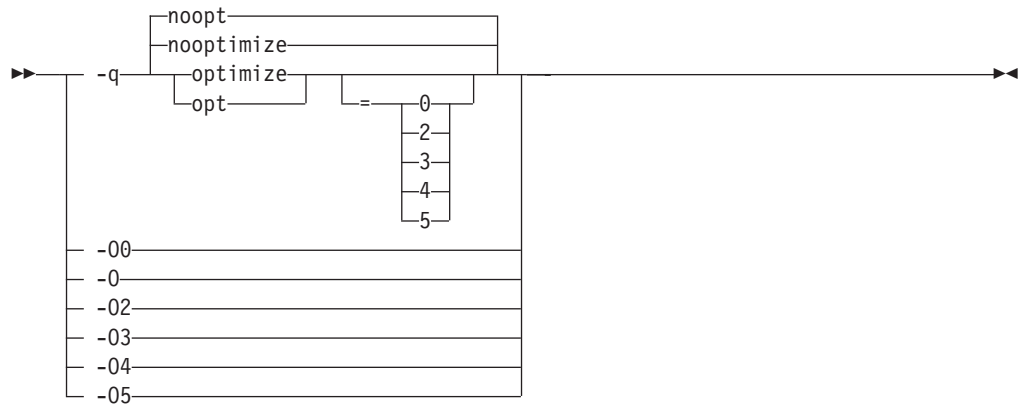
Defaults

-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

Chapter 3. Compiler options reference   **205**

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=fltint: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**.

Aggressive optimizations do *not* include the following floating-point suboptions: **-qfloat=hsflt** | **hssngl**, or anything else that affects the precision mode of a program.

Integer divide instructions are considered too dangerous to optimize even at **-O3**.

Refer to "-qflttrap" on page 119 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 132
- "-qipa" on page 149
- "-qpdf1, -qpdf2" on page 215
- opt_strict
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*.

-qobjmodel (C++ only)
Category

Portability and migration

Pragma equivalent

#pragma object_model

Purpose

Sets the object model to be used for structures, unions, and classes.

The object models differ in the following areas:
- Layout for the virtual function table
- Virtual base class support
- Name mangling scheme

Syntax

Option syntax

```
►►─── -q─objmodel─── = ─┬─classic─┬──────────────────────────────────────────►◄
                        └─ibm─────┘
```

Pragma syntax

```
►►───#pragma─object_model─(─┬─classic─┬───────────────────────────────────────►◄
                            ├─ibm─────┤
                            └─pop─────┘
                            └─)───────┘
```

Defaults

-qobjmodel=classic

Parameters

classic
Uses the object model compatible with V3.6 of the IBM C++ Compiler. This suboption can also be specified using the legacy suboption name of

-qobjmodel=compat, but support for this legacy suboption name may be removed in future releases of the compiler.

ibm

Uses the object model introduced with VisualAge C++ V5.0. Objects compiled with this object model will use less memory and have better performance for deep inheritance with virtual bases.

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

All classes in the same inheritance hierarchy must have the same object model.

Predefined macros

- __OBJECT_MODEL_CLASSIC__ is predefined to 1 when **-qobjmodel=classic** or **#pragma object_model(classic)** is in effect (the default); otherwise, it is undefined.
- __OBJECT_MODEL_IBM__ is predefined to 1 when **-qobjmodel=ibm** or **#pragma object_model(ibm)** is in effect; otherwise, it is undefined.

Examples

To compile myprogram.C with the **ibm** object model, enter:

```
xlc++ myprogram.C -qobjmodel=ibm
```

-qoldpassbyvalue (C++ only)
Category

Portability and migration

Pragma equivalent

#pragma pass_by_value

Purpose

Specifies how classes containing const or reference members are passed in function arguments.

The IBM C++ Compiler V3.6 uses pass by value only if the class has no const or reference data members, and the copy constructor is trivial and the destructor is trivial. VisualAge C++ V5.0 and later compilers use pass by value if the copy constructor is trivial and the destructor is trivial, regardless of const or reference data members. When **-qoldpassbyvalue** is in effect, the compiler uses the 3.6 behavior so that when a class containing a const or reference member is passed as a function argument, it is not passed by value. When **-qnooldpassbyvalue** is in effect, the compiler uses the new behavior so that when a class containing a const or reference member is passed as a function argument, it is passed by value.

The **#pragma pass_by_value** directive allows you greater control over this option for specific files or sections of source code.

Syntax

Option syntax

```
          ┌─nooldpassbyvalue─┐
►►── -q───┴─oldpassbyvalue───┴─────────────────────────────────────────────────◄◄
```

```
►►─┤ Pragma syntax ├─────────────────────────────────────────────────────────────◄◄
```

Pragma syntax

```
├──#──pragma──pass_by_value──(──┬─compat──┬──)─────────────────────────────────────┤
                                ├─ansi────┤
                                ├─default─┤
                                ├─source──┤
                                ├─pop─────┤
                                └─reset───┘
```

Defaults

-qnooldpassbyvalue

Parameters

compat (pragma only)
> Sets **-qoldpassbyvalue** for the code that follows it. This instructs the compiler to use the earlier behavior: when a class containing a `const` or reference member is passed as a function argument, it is not passed by value.

ansi | default (pragma only)
> Sets **-qnooldpassbyvalue** for the code that follows it. This instructs the compiler to use the new behavior: when a class containing a `const` or reference member is passed as a function argument, it is passed by value.

source (pragma only)
> Reverts to the setting specified by the command-line option; if no option has been specified, reverts to the default setting (**-qnooldpassbyvalue**).

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

Use this option if you are linking to libraries compiled with IBM C++ Compiler V3.6 or earlier. Otherwise, functions that have `const` or reference class parameter types will give incorrect behavior when they are called from modules compiled with a later version of the compiler. Library headers compiled with IBM C++ Compiler V3.6 or earlier should be protected with the **#pragma pass_by_value** directive so that users of these libraries will get the correct calling convention for functions in those libraries that use class parameters.

Predefined macros

None.

-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 204
- "-qhot" on page 132
- "-qipa" on page 149
- "-qpdf1, -qpdf2" on page 215
- "-qsmp" on page 247
- "-g" on page 126
- "-qlinedebug" on page 181

-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 a program compiled with **-p**, and it ends normally, it writes the recorded information to a mon.out file; a program compiled with **-pg** writes a gmon.out file. You can then use the **prof** or **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 -cxlc
 myprogram.o -p -o program
```

Related information

- "-qtbtable" on page 270
- See your operating system documentation for more information on the **prof** and **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 80
- "-E" on page 101
- "-qppline" on page 222
- "-qsyntaxonly (C only)" on page 267

-qpath

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. This option is preferred over the **-B** and **-t** options.

Syntax

```
►►── -q─path──=──┬─a─┬──:─directory_path────────────────────────────────►◄
                 ├─b─┤
                 ├─c─┤
                 ├─C─┤
                 ├─d─┤
                 ├─E─┤
                 ├─f─┤
                 ├─I─┤
                 ├─L─┤
                 ├─l─┤
                 ├─m─┤
                 └─p─┘
```
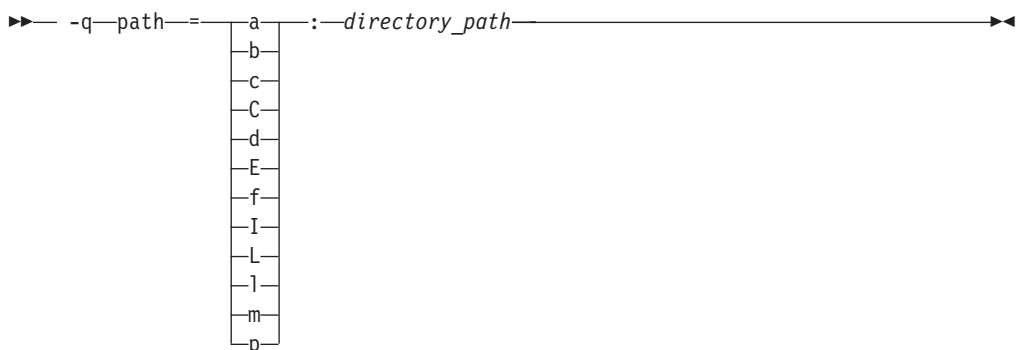
Defaults

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

Parameter	Description	Executable name
c	Compiler front end	xlcentry, xlCentry
C++ C	C++ compiler front end	xlCentry
d	Disassembler	dis
E	**CreateExportList** utility	CreateExportList
C++ f	**c++filt** utility	c++filt
I	High-level optimizer, compile step	ipa
L	High-level optimizer, link step	ipa
l	Linker	ld
C++ m	Linkage helper	munch
p	Preprocessor	n/a

Usage

The **-qpath** option overrides the **-F**, **-t**, and **-B** options.

Note that using the **p** suboption causes 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 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 76
- "-F" on page 110
- "-t" on page 268

-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, this 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/or link or relink the application with **-qpdf2** and a minimum optimization level of **-O2**, which fine-tunes the optimizations applied according to the profile data collected during the program execution.

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─┘
                   └─pdf2────────────────────────┘
                            └─=──pdfname──=──file_path─┘
```
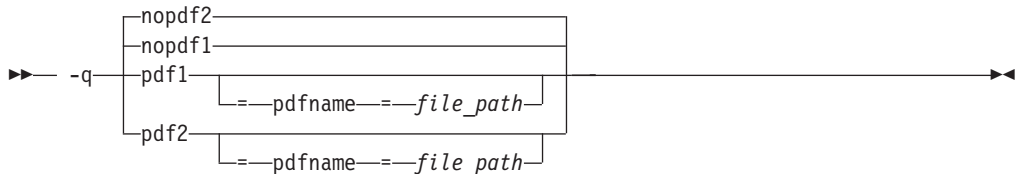
Defaults

-qnopdf1, -qnopdf2

Parameters

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.

You must use the same compilation options for the PDF2 compilation as for the PDF1 compilation.

If you do not want the optimized object files to be relinked during the second step, specify **-qpdf2 -qnoipa**. Note, however, that if you change a source file that was compiled previously with **-qpdf1**, you will need to go through the entire first pass process again.

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. 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*.

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 246 for more information.

For recommended procedures for using PDF, see "Using profile-directed feedback" in the *XL C/C++ Optimization and Programming Guide* in the *XL C/C++ Optimization and Programming Guide*.

The following utility programs, found in /usr/vacpp/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─┬────┬─┬────┬──►◄
             └─-r─scaling─┘                       └─-n─┘ └─-v─┘
```

> 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
```

Related information
- "-qshowpdf" on page 246
- "-qipa" on page 149
- "Optimizing your applications" 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

Defaults

- -qpic=small
- **-qpic=small** when the **-G** or **-qmkshrobj** compiler option is specified.

Parameters

small
> Instructs the compiler to assume that the size of the Table of Contents is no larger than 64 Kb.

large
> Allows the Table of Contents 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

Specifying **-qpic=large** has the same effect as passing **-bbigtoc** to **ld**.

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 58
- "-G" on page 127
- "-qmkshrobj" on page 196

-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 101
- "-P" on page 213

-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

```
►►── -q──┬─prefetch───┬──────────────────────────────────────────────►◄
         └─noprefetch─┘
```

Defaults

-qprefetch

Usage

The **-qnoprefetch** option will not prevent built-in functions such as __prefetch_by_stream from generating prefetch instructions.

Predefined macros

None.

-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 0.

Parameters

number
> An integer literal in the range of -2 147 482 624 to 2147483647. A lower value indicates a higher priority; a higher value indicates a lower priority. Numbers from -214 783 648 to -214 782 623 are reserved for system use. If you do not specify a *number*, the compiler assumes 0.

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. Refer to "Initializing static objects in libraries" in the *XL C/C++ Optimization and Programming Guide* for further discussions on techniques used in handling static object initialization across modules.

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
- "-qmkshrobj" on page 196
- "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
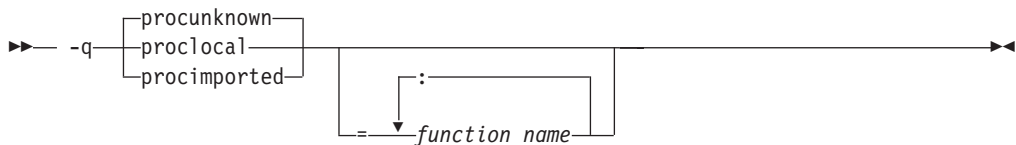
Defaults

-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. You can also use can the **c++filt** utility provided by the compiler for a side-by-side listing of source names and mangled names; see "Demangling compiled C++ names" in the *XL C/C++ Optimization and Programming Guide* for details. (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 93

-qproto (C only)
Category

Object code control

Pragma equivalent
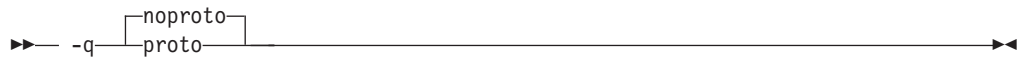
#pragma options [no]proto

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 only 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
```

-Q, -qinline
Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Attempts to inline functions instead of generating calls to those functions, for improved performance.

> **C++** Specifying **-Q** (or **-qinline**) enables automatic inlining by the compiler front end. Specifying **-Q** 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 **-O** along with **-Q** (or **-qinline**) to enable inlining of functions, including those declared with the `inline` specifier. You can also use the **-Q** option to specify restrictions on the functions that should or should not be inlined.

In all cases where **-Q** (or **-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 **-Q!** (or **-qnoinline**) disables all inlining, including that performed by the high-level optimizer with the **-qipa** option, and functions declared explicitly as `inline`.

Syntax

-qinline and -Q syntax — C

```
        ┌─-qnoinline─┐
        │ └─-Q!─────┘                                      │
►►──────┼─-qinline─┐──────────────────────────────────────────────────►◄
          └─-Q────┘
                    └─=size_limit─┐
                                  ┌─:─┐
                                  ▼   │
                    └─ + ─┬───────names─┘
                          └─ - ─┘
```

-qinline and -Q syntax — C++

```
        ┌─-qnoinline─┐
        │ └─-Q!─────┘     │
►►──────┼─-qinline─┐──────────────────────────────────────────────────►◄
          └─-Q────┘
```
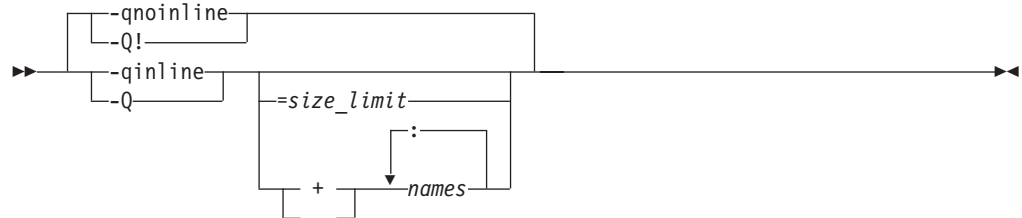
Defaults

-qnoinline or -Q!

Parameters

> **C** *size_limit*

A positive integer representing the number of executable statements in a function. Declarations are not counted, as you can see in the example below:

```
increment()
{
  int a, b, i;
   for (i=0; i<10; i++) /* statement 1 */
   {
```

```
        a=i;                /* statement 2 */
        b=i;                /* statement 3 */
    }
}
```

The number of executable statements in a function must be fewer than or equal to *size_limit* for it to be considered for inlining. Specifying a value of **0** causes no functions to be inlined except those functions listed in the *name* suboption, or those marked with supported forms of the `inline` function specifier. If you do not specify *size*, the default value is 20.

▶   **C**   **+**

The compiler attempts to inline all functions that meet the criterion specified by *size*, as well as those listed by *name*.

▶   **C**   **-**

The compiler attempts to inline all functions that meet the criterion specified by *size*, except those listed by *name*.

▶   **C**   *name*

The name of a function to be inlined. Separate each function name with a colon (**:**). This suboption overrides any setting of the *size* value. Note that this suboption does not affect functions explicitly declared with the `inline` specifier; when **-O** and **-Q | -qinline** are in effect, those functions are *always* considered for inlining. You can specify this suboption as an argument to both the positive and negative forms of the options, to precisely control which functions are most likely to be inlined.

A warning message is issued for functions that are not defined in the source file being compiled.

Usage

To maximize inlining, specify optimization (**-O**) and also specify the appropriate **-qinline** or **-Q** 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 -O -qnoinline
```

▶   **C**   To compile `myprogram.c` so that the compiler attempts to inline all functions of fewer than 12 statements, enter:

```
xlc myprogram.c -O -qinline=12
```

▶ C ▌ Assuming that the functions `salary`, `taxes`, `expenses`, and `benefits` have more than 20 executable statements each, to compile `myprogram.c` so that the compiler attempts to inline all appropriate functions (that is, those that have fewer than the default of 20 statements) *plus* these functions, enter:

```
xlc myprogram.c -O -qinline+salary:taxes:expenses:benefits
```

▶ C ▌ Assuming that the functions `salary`, `taxes`, `expenses`, and `benefits` have fewer than 20 executable statements each, to compile `myprogram.c` so that the compiler attempts to inline all appropriate functions (that is, those that have fewer than the default of 20 statements) *except* these functions, enter:

```
xlc myprogram.c -O -qinline-salary:taxes:expenses:benefits
```

▶ C ▌ You can use a size value of zero along with function names to inline specific functions. For example:

```
-O -qinline=0
```

followed by:

```
-qinline+salary:taxes:benefits
```

causes *only* the functions named `salary`, `taxes`, or `benefits` to be inlined, if possible, and no others.

Related information
- "-g" on page 126
- "-qipa" on page 149
- "-O, -qoptimize" on page 204
- "The inline function specifier" in the *XL C/C++ Language Reference*

-r

Category

Object code control

Pragma equivalent

None.

Purpose

Produces a relocatable object, even though the file contains unresolved symbols.

Syntax

▶▶── -r ── ▶◀

Defaults

Not applicable.

Usage

A file produced with this flag is expected to be used as a file parameter in another compiler invocation.

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 164

-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 and/or optimized. The report also includes diagnostic information to show why specific loops could not be parallelized and/or vectorized. For instance, when used with **-qhot=simd** and **-qenablevmx**, messages are provided to identify non-stride-one references that may 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.

When used with **-qipa=clonearch**, **-qreport** produces transformation reports on the procedures that are cloned for the architectures specified by the option.

Syntax

```
►►─ -q─┬─────────┬───────────────────────────────────────────►◄
        ├─noreport─┤
        └─report──┘
```

Defaults

-qnoreport

Usage

For **-qreport** to generate a loop transformation listing, you must also specify one of the following options on the command line:

- **-qhot[=simd]**
- **-qsmp**
- **-O5**
- **-qipa=level=2**

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**

For **-qreport** to generate a function cloning listing, you must also specify **-qipa=clonearch**.

If you use **-qreport** with **-O5** or **-qipa=level=2**, the report will be generated after the link step.

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.c
```

Related information

* "-qhot" on page 132
* "-qipa" on page 149
* "-qsmp" on page 247
* "-qoptdebug" on page 211
* "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*

-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
```

-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 235
- "-qroptr"

-qroptr

Category

Object code control

Pragma equivalent

None.

Purpose

Specifies the storage location for constant pointers.

When **-qroptr** is in effect, constant pointers, virtual function tables, and virtual type tables are placed in read-only storage. When **-qnoroptr** is in effect, pointers, virtual function tables, and virtual type tables are placed are placed in read/write storage.

Syntax

```
           ┌─noroptr─┐
►►── -q─────┤         ├──────────────────────────────────────────────►◄
           └─roptr───┘
```

Defaults

-qnoroptr

Usage

A constant pointer is equivalent to an address constant. For example:
```
int* const p = &n;
```

When **-qnoroptr** is in effect, you can change the values of constant pointers, virtual function tables, and virtual type tables without generating errors.

The **-qroptr** can improve runtime performance, save storage, and provide shared access, but code that attempts to modify a read-only constant value generates a memory error. For example, assume the following code, which attempts to change the address that c1_ptr points to:
```
char c1 = 10;
char c2 = 20;
char* const c1_ptr = &c1;

int main() {
    *(char**)&c1_ptr = &c2;
}
```

Compiling this code with the **-qroptr** option specified will result in a segmentation fault at run time.

You should not use **-qroptr** for compiled code that will become part of a shared library.

Predefined macros

None.

Related information
- "-qro" on page 235
- "-qroconst" on page 237

-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

```
                 ┌─nortti───────┐
►►── -q──┴─rtti──=──┬─all────────┬─┴──────────────────────────►◄
                    ├─type───────┤
                    ├─typeinfo───┤
                    ├─dyna───────┤
                    └─dynamiccast─┘
```
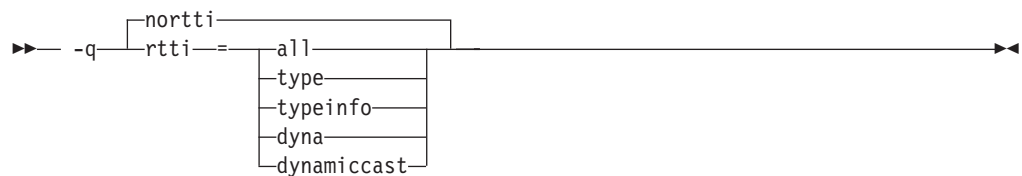
Defaults

-qnortti

Parameters

all The compiler generates the information needed for the RTTI `typeid` and `dynamic_cast` operators.

type | typeinfo
The compiler generates the information needed for the RTTI `typeid` operator, but the information needed for `dynamic_cast` operator is not generated.

dyna | dynamiccast
The compiler generates the information needed for the RTTI `dynamic_cast` operator, but the information needed for `typeid` operator is not generated.

Specifying **-qrtti** with no suboptions is equivalent to **-qrtti=all**.

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** or **-qrtti=all** is in effect; otherwise, it is undefined.
- __RTTI_DYNAMIC_CAST__ is defined to 1 when **-qrtti** or **-qrtti=all** | **dynamiccast** is in effect; otherwise, it is undefined.
- __RTTI_TYPE_INFO__ is defined to 1 when **-qrtti** or **-qrtti=all** | **typeinfo** 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 103

-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 126

-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 101
- "-P" on page 213

-qsaveopt
Category

Object code control

Pragma equivalent

None.

Purpose

Saves the command-line options used for compiling a source file, 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 may 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──┘
```
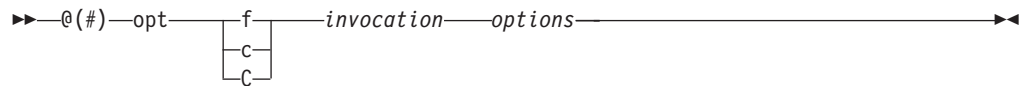
where:

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 spaces.

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:
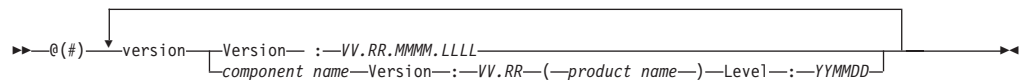
```
      ┌────────────────────────────────────────────────────────────┐
►►──@(#)──┴──version──┬──Version─ :──VV.RR.MMMM.LLLL─────────────────┴──►◄
                      └──component_name──Version──:──VV.RR──(──product_name──)──Level──:──YYMMDD──┘
```

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 (PTF). 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 **what** command on the resulting t.o object file produces information
similar to the following:

```
opt c /usr/vac/bin/xlc t.f -c -qsaveopt -qhot
version IBM XL C/C++ for AIX, V10.1
version Version: 10.01.0000.0000
version Driver Version: 10.01(C/C++) Level: YYMMDD
version Front End  Version: 10.01(C/C++)  Level: YYMMDD
version C Front End Version : 10.01(C/C++)  Level: YYMMDD
version High Level Optimizer Version: 10.01(C) and 12.01(Fortran)  Level: YYMMDD
version Low Level Optimizer  Version: 10.01(C) and 12.01(Fortran) Level: YYMMDD
```

In the first line, `c` identifies the source used as C, `/usr/vacpp/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 (PTFs) you have installed on your system.

Related information
- "-qversion" on page 296

-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
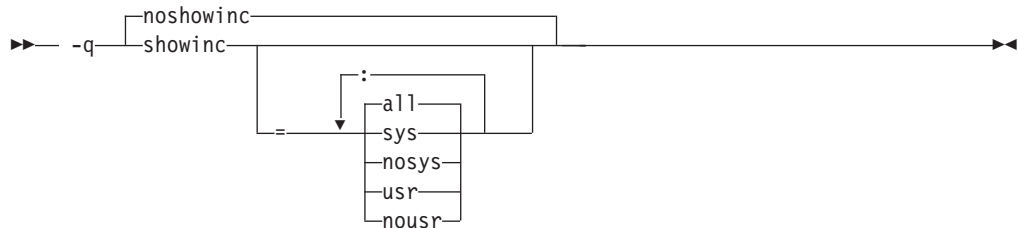
Defaults

-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 251

-qshowmacros
Category

"Output control" on page 39

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───┘
```
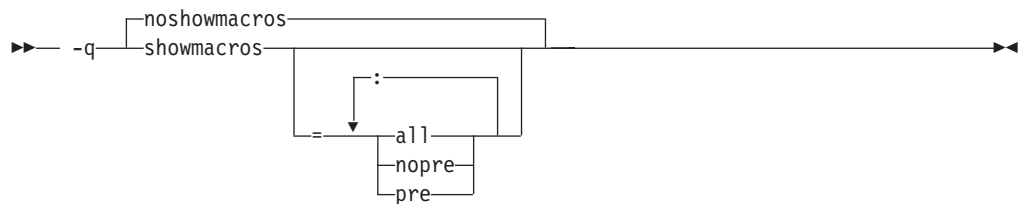
Defaults

-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 101
- "-P" on page 213

-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

```
►►─── -q ──┬─noshowpdf─┬──────────────────────────────────────────────────►◄
           └─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 215.

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 215
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*

-qsmallstack

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Reduces the size of the stack frame.

Syntax

```
        ┌─nosmallstack─┐
►►── -q──┴─smallstack──┴──────────────────────────────────►◄
```

Defaults

-qnosmallstack

Usage

AIX limits the stack size to 256 MB. 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 126
- "-qipa" on page 149

-qsmp

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Enables parallelization of program code.

Syntax
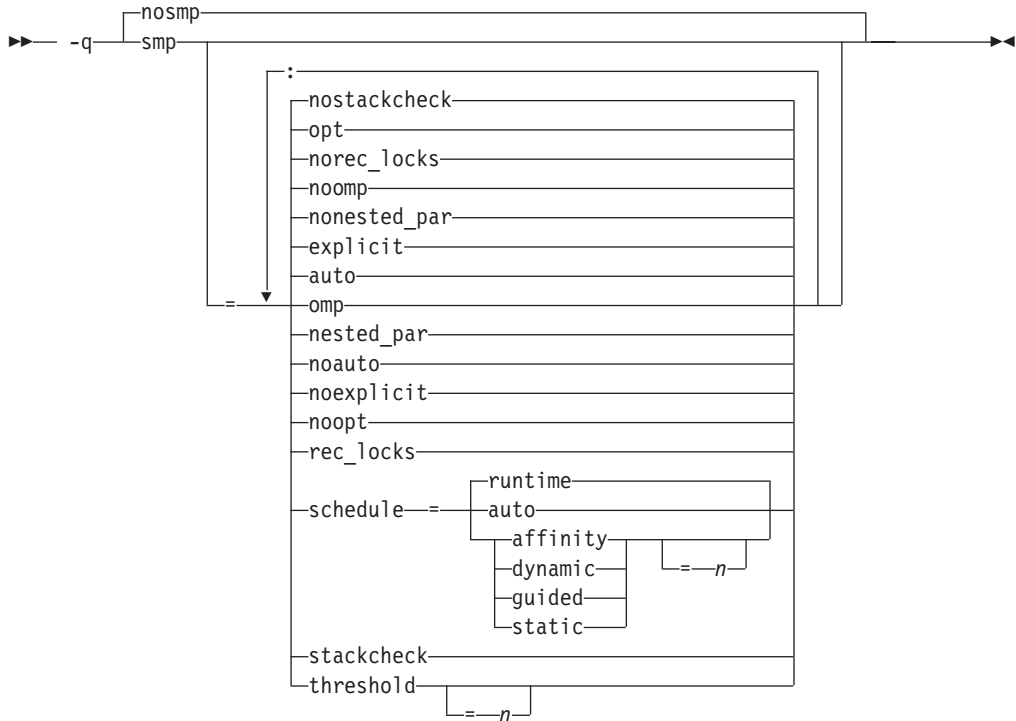
Defaults

-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 SMP or 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 that 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.

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
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. Chunk size (n) must<u>not</u> be specified when

using **auto**. If chunk size (*n*) is specified, the compiler issues a severe error message. Note that when both the option **-qsmp=schedule** and *OMP_SCHEDULE* are used, the option will override the environment variable.

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`

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:ibm** 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**.

Predefined macros

> C   When **-qsmp** is in effect, _IBMSMP is predefined to a value of 1, which indicates that IBM SMP directives are recognized; otherwise, it is not defined.

Related information

- "-O, -qoptimize" on page 204
- "-qthreaded" on page 276
- "Environment variables for parallel processing" on page 25
- "Pragma directives for parallel processing" on page 359
- "Built-in functions for parallel processing" on page 441

-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 20.

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 182
- "-qlistopt" on page 184
- "-qprint" on page 223

-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

```
►►── -q─sourcetype──=──┬─default──────────┬──────────────────────────────►◄
                       ├─assembler────────┤
                       ├─assembler-with-cpp─┤
                       ├─c────────────────┤
                       └─c++──────────────┘
```
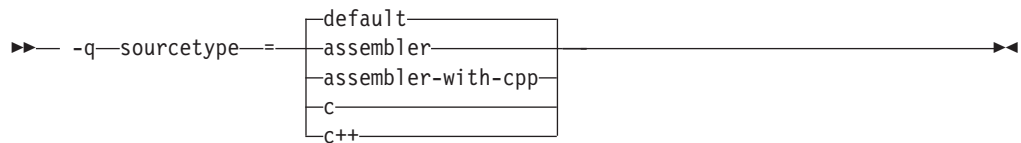
Defaults

-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 56

-qspeculateabsolutes
Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Works with the -qtocmerge -bl:file for non-IPA links and with the -bl:file for IPA links to disable speculation at absolute addresses.

The bl:file is necessary for the compiler to know which addresses are absolutes.

Syntax

```
         ┌─speculateabsolutes───┐
►►── -q───┼─nospeculateabsolutes─┤────────────────────────────────────────►◄
```

Defaults

-qspeculateabsolutes

Predefined macros

None.

Related information
- "-qtocmerge" on page 282

-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

```
►►── -q──┬─nosrcmsg─┬──────────────────────────────────────────────────────────►◄
         └─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
```

-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

```
►►── -q─┬─nostaticinline─┬───────────────────────────────────────────────►◄
        └──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*
- "-qkeepinlines (C++ only)" on page 160

-qstaticlink

Category

Linking

Pragma equivalent

None.

Purpose

Controls how shared and non-shared runtime libraries are linked into an application.

When **-qstaticlink** is in effect, the compiler links only static libraries with the object file named in the invocation. When **-qnostaticlink** is in effect, the compiler links shared libraries with the object file named in the invocation.

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─┘
```
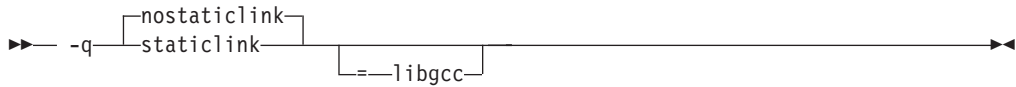
Defaults

-qnostaticlink

Parameters

libgcc
When specified together with **nostaticlink**, the compiler links to the shared version of libgcc. When specified together with **staticlink**, the compiler links to the static version of libgcc.

Usage

The following table shows the equivalent GNU and XL C/C++ options for specifying linkage of shared and non-shared libraries.

Table 25. Option mappings: control of the GNU linker

GNU option	Meaning	XL C/C++ 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=libgcc
-static-libgcc	Link with the static version of libgcc.	-qstaticlink=libgcc

WARNING: 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 you compile. IBM strongly recommends that you seek legal advice before using this option.

Predefined macros

None.

-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 /usr/vacpp/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 /usr/vacpp/include/) or by the **-qcpp_stdinc** option
- The directory specified in the configuration file for the system header files (this is normally /usr/include/), or by the **-qc_stdinc** and **-qcpp_stdinc** optionsor 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

```
►►── -q ──┬─stdinc──┬──────────────────────────────────────────────────►◄
          └─nostdinc─┘
```

Defaults

-qstdinc

Usage

The search order of header files is described in "Directory search sequence for include files" on page 13.

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 90
- "-qcpp_stdinc (C++ only)" on page 91
- "-I" on page 135
- "Directory search sequence for include files" on page 13

-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

Option syntax

```
►►── -q ──┬─nostrict──────────────────────────────────────────────────────►◄
          └─strict──┐
                    │                  ┌─── : ───┐
                    └─ = ──┬─►─┬─all──────────────┬─┐
                              │ ├─none─────────────┤ │
                              │ ├─precision────────┤ │
                              │ ├─noprecision──────┤ │
                              │ ├─exceptions───────┤ │
                              │ ├─noexceptions─────┤ │
                              │ ├─ieeefp───────────┤ │
                              │ ├─noieeefp─────────┤ │
                              │ ├─nans─────────────┤ │
                              │ ├─nonans───────────┤ │
                              │ ├─infinities───────┤ │
                              │ ├─noinfinities─────┤ │
                              │ ├─subnormals───────┤ │
                              │ ├─nosubnormals─────┤ │
                              │ ├─zerosigns────────┤ │
                              │ ├─nozerosigns──────┤ │
                              │ ├─operationprecision┤ │
                              │ ├─nooperationprecision┤│
                              │ ├─order────────────┤ │
                              │ ├─noorder──────────┤ │
                              │ ├─association──────┤ │
                              │ ├─noassociation────┤ │
                              │ ├─reductionorder───┤ │
                              │ ├─noreductionorder─┤ │
                              │ ├─guards───────────┤ │
                              │ ├─noguards─────────┤ │
                              │ ├─library──────────┤ │
                              │ └─nolibrary────────┘
```
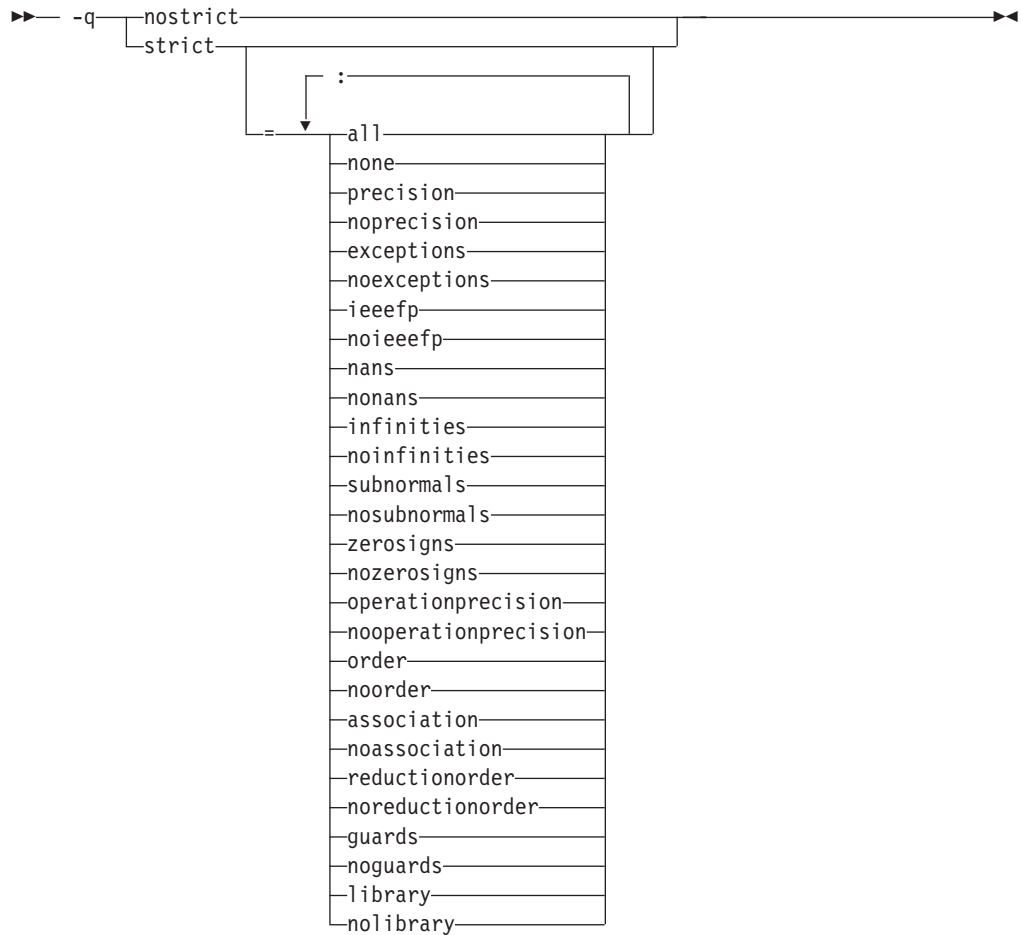
Defaults

- 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**, **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**, 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 signaling 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.

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 these reductions.

guards | noguards

    guards disables moving operations past guards (that is, past **IF** statements**ifs**

or out of loops) or calls which control whether the operation should be executed or not. enables these moving operations.

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[=subtopions]** or **-qnostrict** combinations sets **-qfloat** suboptions as follows:
* **-qstrict** or **-qstrict=all** sets **-qfloat=nofltint:norsqrt:rngchk**. **-qnostrict** or **-qstrict=none** sets **-qfloat=fltint: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**.

To override any of these settings, specify the appropriate **-qfloat** suboptions after **-q[no]strict[=suboption_list]** on the command line.

Predefined macros

None.

Examples

To compile myprogram.c so that the aggressive optimizations of **-O3** are turned off, range checking is turned off (**-qfloat=fltint**), 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=fltint: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
- "-qfloat" on page 114
- "-qhot" on page 132
- "-O, -qoptimize" on page 204

-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 204

-qsuppress
Category

Listings, messages, and compiler information

Pragma equivalent

"#pragma report (C++ only)" on page 350

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 ─┴─┬──────────────►◄
```
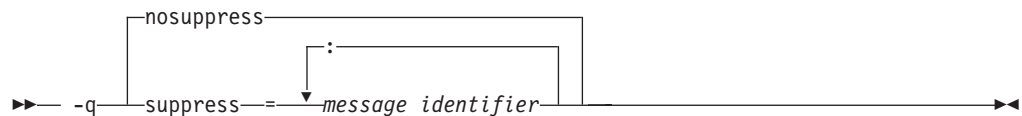
Defaults

-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 18 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 113

-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 126
- "-qdbxextra (C only)" on page 95
- "-qstatsym" on page 258

-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 80
- "-c" on page 79
- "-E" on page 101
- "-o" on page 203
- "-P" on page 213

-t

Category

Compiler customization

Pragma equivalent

None.

Purpose

Applies the prefix specified by the **-B** option to the designated components.

Syntax

```
►►──  -t ──┬── a ──┬──────────────────────────────────────────────────────►◄
           ├── b ──┤
           ├── c ──┤
           ├── C ──┤
           ├── d ──┤
           ├── E ──┤
           ├── f ──┤
           ├── I ──┤
           ├── L ──┤
           ├── l ──┤
           ├── m ──┤
           └── p ──┘
```

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
E	**CreateExportList** utility	CreateExportList
▶ C++ ◀ f	**c++filt** utility	c++filt
I	High-level optimizer, compile step	ipa
L	High-level optimizer, link step	ipa
l	Linker	ld
▶ C++ ◀ m	Linkage helper	munch
p	Preprocessor	n/a

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 that using the **p** suboption causes 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 76

-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

<u>full</u>
> 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
- "-g" on page 126

-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 ────┘
```
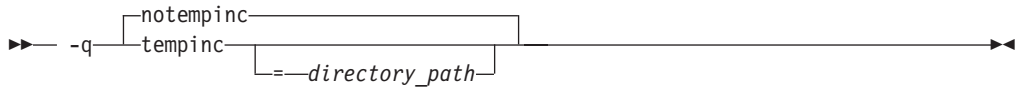
Defaults

-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 328
- "-qtmplinst (C++ only)" on page 280
- "-qtemplateregistry (C++ only)" on page 274
- "-qtemplaterecompile (C++ only)" on page 273
- "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 280
- "-qtempinc (C++ only)" on page 271
- "-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

```
              ┌─notemplateregistry──────────────┐
►►── -q───────┼─templateregistry────────────────┤─────────────────►◄
                                └─=─file_path─┘
```

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 280
- "-qtempinc (C++ only)" on page 271
- "-qtemplaterecompile (C++ only)" on page 273
- "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 271
- "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

```
          ┌─nothreaded─┐
►►── -q───┴─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 247

-qtimestamps
Category

"Output control" on page 39

Pragma equivalent

none.

Purpose

Controls whether or not implicit time stamps are inserted into an object file.

Syntax

```
      ┌─timestamps──┐
►►── -q─┴─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.

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.

Note: This option is only supported on AIX for POWER version 5.3 with the 5300-05 Technology Level and higher.

Syntax

```
►►── -q──┬─tls─┬────────────────────────────┬──────────────────────────►◄
          │     ┌────unsupported────┐        │
          └─=──┼────default────────┤         │
          │     ├─global-dynamic───┤         │
          │     ├─initial-exec─────┤         │
          │     ├─local-exec───────┤         │
          │     └─local-dynamic───┘          │
          └─notls────────────────────────────┘
```
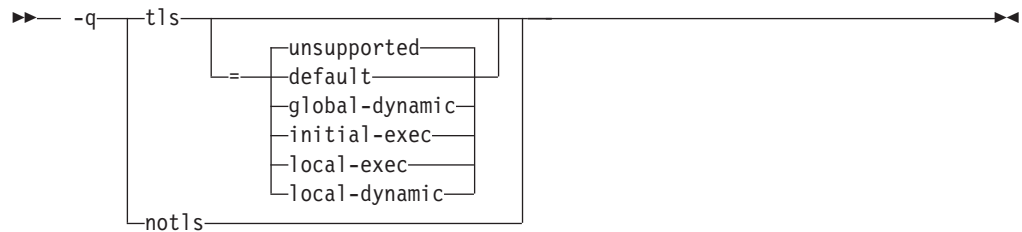
Defaults

-qtls=unsupported

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)(**-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 221
- "The __thread storage class specifier" in the *XL C/C++ Language Reference*

Chapter 3. Compiler options reference   **279**

-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 321.

Predefined macros

None.

Related information

- "-qtemplateregistry (C++ only)" on page 274
- "-qtempinc (C++ only)" on page 271
- "#pragma do_not_instantiate (C++ only)" on page 321
- "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 93.

-qtocmerge

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Enables TOC merging to reduce TOC pointer loads and improves the scheduling of external loads.

Syntax

```
          ┌─notocmerge─┐
►►── -q────┼─tocmerge───┼──────────────────────────────────────►◄
```

Defaults

-qnotocmerge

Usage

To use **-qtocmerge**, you must also use the **-bImportfile** linker option to specify the name of the file from which the compiler reads.

Predefined macros

None.

-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

```
         ┌─trigraph──┐
►►── -q───┴─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 97
- "-qlanglvl" on page 165

-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

```
►►── -q─tune──=──┬─balanced─┬──────────────────────────────────►◄
                 ├─auto─────┤
                 ├─ppc──────┤
                 ├─ppc970───┤
                 ├─pwr3─────┤
                 ├─pwr4─────┤
                 ├─403──────┤
                 ├─604──────┤
                 ├─pwr5─────┤
                 ├─pwr6─────┤
                 ├─rs64a────┤
                 ├─rs64b────┤
                 └─rs64c────┘
```
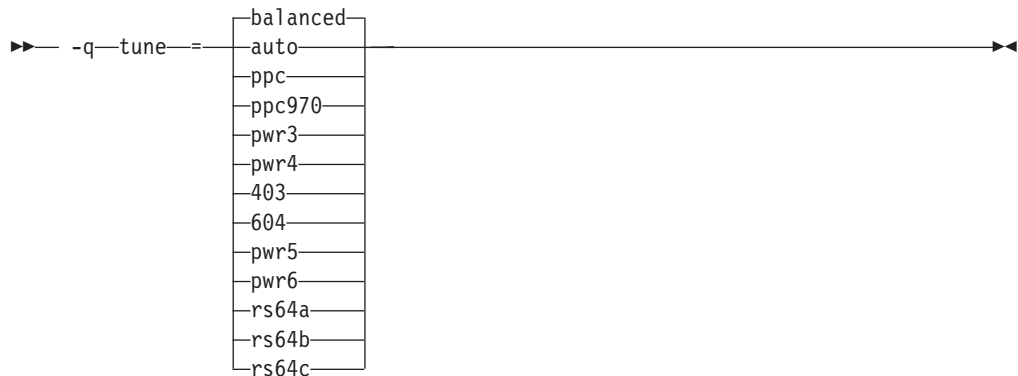
Defaults

-qtune=balanced when the default **-qarch** setting is in effect. Otherwise, the default depends on the effective **-qarch** setting. See Table 26 on page 285 for details.

Parameters

403
> Optimizations are tuned for the PowerPC 403 processor.

604
> Optimizations are tuned for the PowerPC 604 processor.

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.

rs64a
> Optimizations are tuned for the RS64I processor.

rs64b
> Optimizations are tuned for the RS64II processor.

rs64c
> Optimizations are tuned for the RS64III processor.

Note: As of the V9.0 release of the compiler, suboptions representing 601, 602, 603, POWER and POWER2 architectures are deprecated.

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 26. Acceptable **-qarch/-qtune** *combinations*

-qarch option	Default -qtune setting	Available -qtune settings
403	403	auto | 403
604	604	auto | 604
ppc	balanced	auto | 604 | rs64a | rs64b | rs64c | pwr3 | pwr4 | pwr5 | pwr6 | ppc970 | balanced
ppcgr	balanced	auto | 604 | rs64b | rs64c | pwr3 | pwr4 | pwr5 | pwr6 | ppc970 | balanced
ppc64	balanced	autors64a | rs64b | rs64c | pwr3 | pwr4 | pwr5 | pwr6 | ppc970 | balanced
ppc64gr	balanced	auto | rs64b | rs64c | pwr3 | pwr4 | pwr5 | pwr6 | ppc970 | balanced
ppc64grsq	balanced	auto | rs64b | rs64c | pwr3 | pwr4 | pwr5 | pwr6 | ppc970 | balanced
ppc64v	ppc970	auto | ppc970 | pwr6 | balanced
ppc970	ppc970	auto | ppc970 | balanced
pwr3	pwr3	auto | pwr3 | pwr4 | pwr5 | ppc970 | balanced
pwr4	pwr4	auto | pwr4 | pwr5 | ppc970 | balanced
pwr5	pwr5	auto | pwr5 | balanced
pwr5x	pwr5	auto | pwr5 | balanced
pwr6	pwr6	auto | pwr6 | balanced
pwr6e	pwr6	auto | pwr6 | balanced
rs64a	rs64a	auto | rs64a
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 POWER3 hardware platform, enter:

```
xlc -o testing myprogram.c -qtune=pwr3
```

Related information
- "-qarch" on page 67
- "-q32, -q64" on page 58
- "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*

-qtwolink (C++ only)
Category

Linking

Pragma equivalent

None.

Purpose

Minimizes the number of static constructors included from libraries and object files.

When **-qnotwolink** is in effect, all static constructors in .o files and object files are invoked. This generates larger executable files, but ensures that placing a .o file in a library does not change the behavior of a program.

Normally, the compiler links in all static constructors defined anywhere in the object (.o) files and library (.a) files. The **-qtwolink** option makes link time longer, but linking is compatible with older versions of C or C++ compilers.

Syntax

```
             ┌notwolink─┐
►►── -q──────┼─twolink──┼────────────────────────────────►◄
```

Defaults

-qnotwolink

Usage

Before using **-qtwolink**, make sure that any .o files placed in an archive do not change the behavior of the program.

Predefined macros

None.

Examples

Given the include file foo.h:

```
#include <stdio.h>
struct foo {
    foo() {printf ("in foo\n");}
    ~foo() {printf ("in ~foo\n");}
};
```

and the C++ program t.C:

```
#include "foo.h"
foo bar;
```

and the program t2.C:

```
#include "foo.h"
main() { }
```

Compile t.C and t2.C in two steps, first invoking the compiler to produce object files:

```
xlc++ -c t.C t2.C
```

and then link them to produce the executable file a.out:

```
xlc++ t.o t2.o
```

Invoking a.out produces:

```
in foo
in ~foo
```

If you use the AIX **ar** command with the t.o file to produce an archive file t.a:

```
ar rv t.a t.o
```

and then use the default compiler command:

```
xlc++ t2.o t.a
```

the output from the executable file is the same as above:

```
in foo
in ~foo
```

However, if you use the **-qtwolink** option:

```
xlc++ -qtwolink t2.o t.a
```

there is no output from the executable file a.out because the static constructor foo() in t.C is not found.

-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 5, "Compiler predefined macros," on page 379 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 92

-qunique (C++ only)
Category

Object code control

Pragma equivalent

None.

Purpose

Generates unique names for static constructor/destructor file compilation units.

Syntax

```
►►── -q──┬─nounique─┬──────────────────────────────────────────────►◄
         └─unique───┘
```

Defaults

-qnounique

Usage

Unique names are generated with **-qunique** by encoding random numbers into the name of the static constructor and destructor functions. Default behavior is encoding the absolute path name of the source file in the constructor and destructor functions. If the absolute path name will be identical for multiple compilations (for example, if a **make** script is used), the **-qunique** option is necessary.

If you use **-qunique**, you must always link with all .o and .a files. Do not include an executable file on the link step.

Predefined macros

None.

Examples

Suppose you want to compile several files using the same path name, ensuring that static construction works correctly. A makefile may generate the following steps:

```
sqlpreprocess file1.sql > t.C
  xlc++ -qunique t.C -o file1.o
rm -f t.C
sqlpreprocess file2.sql > t.C
  xlc++ -qunique t.C -o file2.o
rm -f t.C
  xlc++ file1.o file2.o
```

Following is a sample makefile for the above example:

```
# rule to get from file.sql to file.o
.SUFFIXES:      .sql
.sql.o:
      sqlpreprocess $< > t.C
      $(CCC) t.C -c $(CCFLAGS) -o $@
      rm -f t.C
```

Related information

- #pragma fini (C only)
- #pragma init (C only)

-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
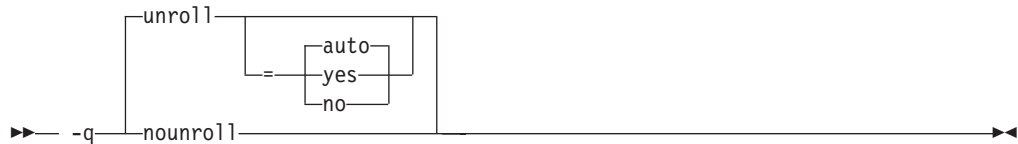
Pragma syntax
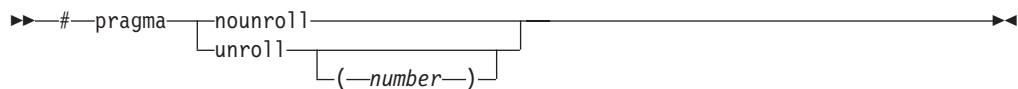
Defaults

-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 315
- "#pragma loopid" on page 331
- "#pragma stream_unroll" on page 352
- "#pragma unrollandfuse" on page 354

-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 103

-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
- "Integral and floating-point promotions" in the *XL C/C++ Language Reference*
- "-qlanglvl" on page 165

-qutf

Category

Language element control

Pragma equivalent

None.

Purpose

Enables recognition of UTF literal syntax.

Syntax

```
►►── -q──┬─noutf─┬──────────────────────────────────────────────────── ►◄
         └─utf───┘
```

Defaults
- ► **C** ◄ -qnoutf
- ► **C++** ◄ **-qutf** for all language levels except **-qlanglvl=strict98**. **-qnoutf** when **-qlanglvl=strict98** is in effect.

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 57

-qvecnvol

Category

Portability and migration

Pragma equivalent

None.

Purpose

Specifies whether to use volatile or non-volatile vector registers.

Volatile vector registers are those whose value is not preserved across function calls or across save context, jump or switch context system library functions. When

-qvecnvol is in effect, the compiler uses both volatile and non-volatile vector registers. When **-qnovecnvol** is in effect, the compiler uses only volatile vector registers.

This option is required for programs where there is risk of interaction between modules built with AIX libraries prior to AIX 5.3 with 5300-03 and vector register use. Restricting the compiler to use only volatile registers will make your vector programs safe but it potentially forces the compiler to store vector data to memory more often and therefore results in reducing performance.

Syntax

```
►►── -q──┬─novecnvol─┬──────────────────────────────────►◄
         └─vecnvol───┘
```

Defaults

-qnovecnvol

Usage

- In order to generate vector-enabled code you should explicitly specify **-qenablevmx** option.
- In order to use the **-qvecnvol** option, you need `bos.adt.include` version 5.3.0.30 or greater installed on your system.
- When **-qnoenablevmx** compiler option is in effect, **-qnovecnvol** option is ignored.
- The **-qnovecnvol** option performs independently from **-qhot=simd | nosimd**, **-qaltivec | -qnoaltivec** and **pragma=nosimd**.
- On AIX 5.3 with 5300-03, by default only 20 volatile registers (vr0-vr19) are used, and 12 non-volatile vector registers (vr20 - vr31) are not used. You can use these registers only when **-qvecnvol** is in effect.
- **-qvecnvol** should be enabled only when no legacy code that saves and restores non-volatile registers is involved. Using **-qvecnvol** and linking with legacy code, may result runtime failure.

Predefined macros

None.

Related information
- "-qaltivec" on page 67
- "-qenablevmx" on page 108
- "-qhot" on page 132

-qversion
Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Displays the version and release of the compiler being invoked.

Syntax

```
►►─── -q ──┬─ noversion ─────────────────┬──────────────────────────────►◄
           └─ 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 AIX, 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 AIX, 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 242

-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,
```

Related information
- "-qflag" on page 113
- "-qsuppress" on page 265

-W

Category

Compiler customization

Pragma equivalent

None.

Purpose

Passes the listed options to a component that is executed during compilation.

Syntax
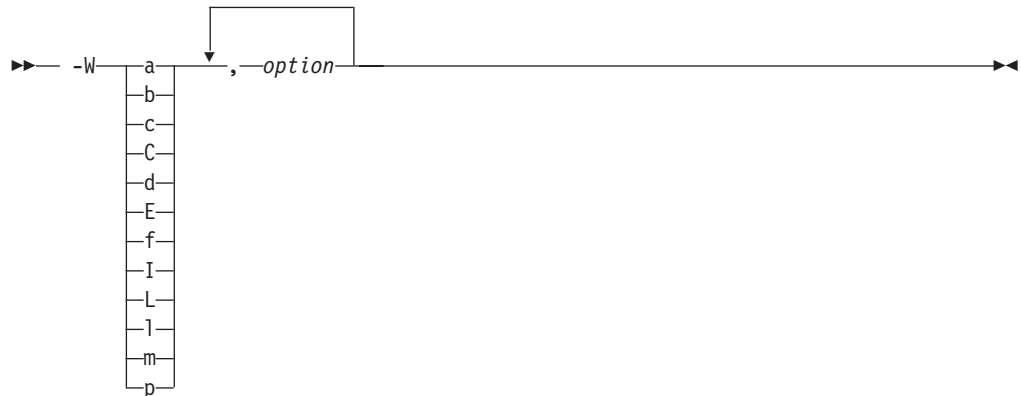
Parameters

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

Parameter	Description	Executable name
b	Low-level optimizer	xlCcode
c	Compiler front end	xlcentry, xlCentry
▶ C++ C	C++ compiler front end	xlCentry
d	Disassembler	dis
E	**CreateExportList** utility	CreateExportList
▶ C++ f	**c++filt** utility	c++filt
I	High-level optimizer, compile step	ipa
L	High-level optimizer, link step	ipa
l	Linker	ld
▶ C++ m	Linkage helper	munch
p	Preprocessor	n/a

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**.

By default, static objects are initialized in the order of priority specified by **#pragma priority** or the "-qpriority (C++ only)" on page 224 option. You can use **-Wm** option to control the initialization order of the objects with the same priorities. Specifying **-Wm -c** instructs the compiler to initialize object files with the same priority in the order in which the files were given on the command line during linking into the library and the static objects within the files are initialized according to their declaration order.**-Wm -r** option, however, specifies that the object files with the same priority are to be initialized in the opposite order in which they were encountered during the linking phase.

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

-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

-qweakexp

Category

Object code control

Pragma equivalent

None.

Purpose

When used with the **-qmkshrobj** or **-G** options, includes or excludes global symbols marked as weak from the export list generated when you create a shared object.

Syntax

```
►►── -q──┬─weakexp───┬──────────────────────────────────────────────►◄
         └─noweakexp─┘
```

Defaults

-qweakexp: weak symbols are exported.

Usage

See "-qweaksymbol" on page 303 for a description of weak symbols.

This option is used together with the **-qmkshrobj** or **-G** options. See the descriptions of "-qmkshrobj" on page 196 or "-G" on page 127 for more information.

Predefined macros

None.

Examples

To compile `myprogram.c` into a shared object, and prevent weak symbols from being exported, enter:

```
xlc myprogram.c -qmkshrobj -qnoweakexp
```

Related information

-qweaksymbol

Category

Object code control

Pragma equivalent

None.

Purpose

Enables the generation of weak symbols.

When **-qweaksymbol** is in effect, the compiler generates weak symbols for the
following:
- Inline functions with external linkage.
- Identifiers specified as weak with **#pragma weak** or __attribute__((weak)).

Syntax

```
           ┌─weaksymbol───┐
►►── -q────┴─noweaksymbol─┴──────────────────────────────────────────────►◄
```

Defaults

-qweaksymbol

Usage

> **C++** When compiling C++ programs containing extern inline functions, you
can use the **-qweaksymbol** compiler option to suppress linker messages warning
of duplicate symbols.

Predefined macros

None.

Related information
- "#pragma weak" on page 356
- "-qweakexp" on page 302
- "The weak variable attribute" and "The weak function attribute" in the *XL C/C++
  Language Reference*

-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 20.

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 27. 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 74
- "#pragma mc_func" on page 335

-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

```
                  ┌─dn──┐
                  ├─n───┤
►►─── -y──────────┼─m───┼──────────────────────────────────────►◄
                  ├─p───┤
                  ├─z───┤
                  ├─di──┤
                  ├─dm──┤
                  ├─dna─┤
                  ├─dnz─┤
                  ├─dp──┤
                  └─dz──┘
```

Defaults

-yn, -ydn

Parameters

The following suboptions are valid for binary floating-point types only:

m Round toward minus infinity.

n̲ Round to the nearest representable number, ties to even.

p Round toward plus infinity.

z Round toward zero.

The following suboptions are valid for decimal floating-point types only:

di Round toward infinities (away from zero).

dm
   Round toward minus infinity.

dn̲ Round to the nearest representable number, ties to even.

dna
   Round to the nearest representable number, ties away from zero.

dnz
   Round to the nearest representable number, ties toward zero.

dp Round toward plus infinity.

dz 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 -ydz
```

-Z

Category

Linking

Pragma equivalent

None.

Purpose

Specifies a prefix for the library search path to be used by the linker.

Syntax

►►── -Z─*string*────────────────────────────────────►◄

Defaults

By default, the linker searches the /usr/lib/ directory for library files.

Parameters

string
   Represents the prefix to be added to the directory search path for library files.

Predefined macros

None.

Chapter 4. Compiler pragmas reference

The following sections describe the pragmas available:

- "Pragma directive syntax"
- "Scope of pragma directives" on page 310
- "Summary of compiler pragmas by functional category" on page 310
- "Individual pragma descriptions" on page 314

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 340.

#pragma *name*

This form uses the following syntax:

```
►►─#─pragma─┬─name─(─suboptions─)─┬──────────────────────►◄
            └◄──────────────────┘
```

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:

```
►►─_Pragma─(─"─┬─name─(─suboptions─)─┬─"─)──────────────►◄
              └◄───────────────────┘
```

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 311
- "C++ template pragmas" on page 311
- "Floating-point and integer control" on page 311
- "Error checking and debugging" on page 311
- "Listings, messages and compiler information" on page 312
- "Optimization and tuning" on page 312
- "Object code control" on page 313
- "Portability and migration" on page 314

For descriptions of these categories, see "Summary of compiler options by functional category" on page 39.

Language element control

Table 28. 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 335	Allows you to embed a short sequence of machine instructions "inline" within your program source code.
"#pragma options" on page 340	Specifies compiler options in your source program.

C++ template pragmas

Table 29. C++ template pragmas

Pragma	Description
"#pragma define, #pragma instantiate (C++ only)" on page 319	Provides an alternative method for explicitly instantiating a template class.
"#pragma do_not_instantiate (C++ only)" on page 321	Prevents the specified template declaration from being instantiated.
"#pragma implementation (C++ only)" on page 328	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 30. 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 31. Error checking and debugging pragmas

Pragma	Description
"#pragma ibm snapshot" on page 327	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.
"#pragma operator_new (C++ only)" on page 339	Determines whether the `new` and `new[]` operators throw an exception if the requested memory cannot be allocated.

Listings, messages and compiler information

Table 32. Listings, messages and compiler information pragmas

Pragma	Description
"#pragma report (C++ only)" on page 350	Controls the generation of diagnostic messages.

Optimization and tuning

Table 33. Optimization and tuning pragmas

Pragma	Description
"#pragma block_loop" on page 315	Marks a block with a scope-unique identifier.
"#pragma STDC cx_limited_range" on page 351	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 320	Lists identifiers that are not aliased to each other within the scope of their use.
"#pragma execution_frequency" on page 322	Marks program source code that you expect will be either very frequently or very infrequently executed.
"#pragma expected_value" on page 324	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 331	Informs the compiler that a named function never returns to the instruction following a call to that function.
"#pragma loopid" on page 331	Marks a block with a scope-unique identifier.
#pragma nosimd	When used with **-qhot=simd**, 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 342	Allows you to specify optimization options at the subprogram level that override optimization options given on the command line.
"#pragma reachable" on page 348	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 349	Specifies registers that may be altered by functions specified by **#pragma mc_func**.
"#pragma stream_unroll" on page 352	When optimization is enabled, breaks a stream contained in a for loop into multiple streams.

Table 33. Optimization and tuning pragmas (continued)

Pragma	Description
#pragma unroll	Controls loop unrolling, for improved performance.
"#pragma unrollandfuse" on page 354	Instructs the compiler to attempt an unroll and fuse operation on nested `for` loops.

Object code control

Table 34. 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 comment" on page 318	Places a comment into the object module.
▶ C "#pragma fini (C only)" on page 325	Specifies the order in which the runtime library calls a list of functions after main() completes or exit() is called.
"#pragma hashome (C++ only)" on page 326	Informs the compiler that the specified class has a home module that will be specified by **#pragma ishome**.
"#pragma ishome (C++ only)" on page 330	Informs the compiler that the specified class's home module is the current compilation unit.
▶ C "#pragma init (C only)" on page 329	Specifies the order in which the runtime library calls a list of functions before main() is called.
"#pragma map" on page 332	Converts all references to an identifier to another, externally defined identifier.
"#pragma pack" on page 344	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 349	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 356	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 35. Portability and migration pragmas

Pragma	Description
#pragma align	Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data.
#pragma namemangling (C++ only)	Chooses the name mangling scheme for external symbol names generated from C++ source code.
"#pragma namemanglingrule (C++ only)" on page 336	Provides fined-grained control over the name mangling scheme in effect for selected portions of source code, specifically with respect to the mangling of cv-qualifiers in function parameters.
#pragma object_model (C++ only) Sets the object model to be used for structures, unions, and classes.	
#pragma pass_by_value (C++ only)	Specifies how classes containing const or reference members are passed in function arguments.

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 309 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 62.

#pragma alloca (C only)

See "-qalloca, -ma (C only)" on page 65.

#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 331
- "-qunroll" on page 290
- "#pragma unrollandfuse" on page 354
- "#pragma stream_unroll" on page 352

#pragma chars
See "-qchars" on page 84.

#pragma comment
Category

Object code control

Purpose

Places a comment into the object module.

Syntax

```
►►──#──pragma──comment──(──┬──compiler─────────────────────────┬──)──►◄
                          ├──date────────────────────────────┤
                          ├──timestamp───────────────────────┤
                          ├──copyright──┐                     │
                          └──user───────┴──┬─────────────────┬┘
                                           └─,─"─token_sequence─"─┘
```
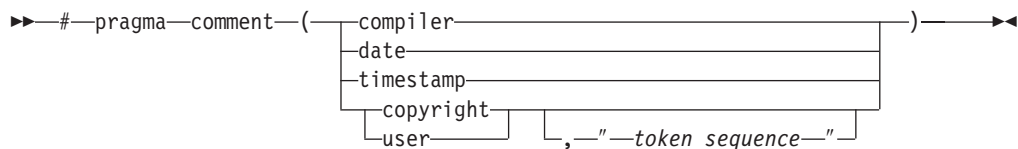
Parameters

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 that following program code is compiled to produce output file `a.out`:

```
#pragma comment(date)
#pragma comment(compiler)
#pragma comment(timestamp)
#pragma comment(copyright,"My copyright")

int main() {

return 0;
}
```

Issuing the command:

```
strings a.out
```

will cause the comment information embedded in `a.out` to be displayed, along with any other strings that may be found in `a.out`. For example, assuming the program code shown above:

```
Mon Mar 1 10:28:03 2008
XL C/C++ for AIX Compiler Version 10.1
Mon Mar 1 10:28:09 2008
My copyright
```

#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 equivalent functionality to standard C++ explicit instantiation, and is provided for backwards compatibility purposes only. New applications should use standard C++ explicit instantiation.

The pragma can appear anywhere an explicit instantiation statement 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 321

#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──┬──)──►◄
      └──┬──┐───┘                 ▲   └──┬──┐───┘                 │
         └─▼─┘                    │      └─▼─┘                    │
            *                     └───────────* ─────────────────┘
```
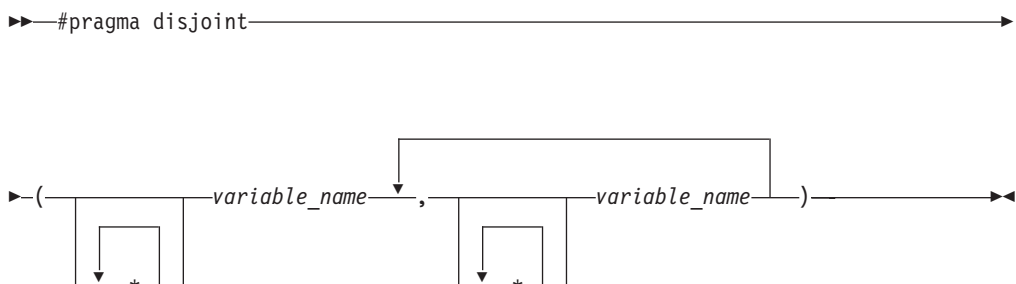
Parameters

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

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.

You can also use the **-qtmplinst** option to suppress implicit instantiation of template declarations for multiple compilation units. See "-qtmplinst (C++ only)" on page 280.

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 319
* "-qtmplinst (C++ only)" on page 280
* "Explicit instantiation" in the *XL C/C++ Language Reference*
* "-qtempinc (C++ only)" on page 271
* "-qtemplateregistry (C++ only)" on page 274

#pragma enum

See "-qenum" on page 104.

#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 322

#pragma fini (C only)
Category

"Object code control" on page 313

Purpose

Specifies the order in which the runtime library calls a list of functions after main() completes or exit() is called.

For shared libraries, the fini functions are called when the shared library is loaded from memory. For example, when using dynamic loading, this happens at the point when dlclose() is called.

Syntax

```
►►─#─pragma─fini─(──┬──────────────┬─)─────────────────────────►◄
                    │      ,        │
                    └─function_name─┘
```

Usage

Any function that is specified in the pragma should have return type void (for example, `void fA();`) and take no parameters. Functions that have a non-void return type are accepted but the return value is discarded.

Functions that take parameters are ignored with a warning since the parameters would contain garbage values.

Within the same compilation unit, the list of functions in `pragma fini` are called in the order specified. Similarly, within the same compilation unit, functions specified in more than one pragma fini are called in the order in which the pragmas are encountered in the source.

In general, the order of static termination across files and across libraries is nonstandard and therefore, a non-portable behavior. It is not advisable to build any dependency on this behavior. The order of functions across files is undefined, even when using the **-Wm** option.

When mixing C and C++ files, the relative order of init or fini functions in C files with respect to the static constructors/destructors in C++ files is undefined. The **-qunique** option can interact with `pragma fini`.

Note: A C++ invocation, such as **xlC** or the redistributable tools **linkxlC** or **makeC++SharedLib** must be used at link time.

Related information
- "#pragma init (C only)" on page 329
- "-qunique (C++ only)" on page 288

#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 330

#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 126
- "-O, -qoptimize" on page 204

#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 271
- "Using C++ templates"

#pragma info

See "-qinfo" on page 141.

#pragma init (C only)
Category

"Object code control" on page 313

Purpose

Specifies the order in which the runtime library calls a list of functions before main() is called.

For shared libraries, the init functions are called when the shared library is loaded to memory. For example, when using dynamic loading, this happens at the point when dlopen() is called.

Syntax

```
►►──#──pragma──init──(──▼──function_name──)────────────────────────►◄
                        └──────,──────┘
```

Usage

Any function that is specified in the pragma should have return type void (for example, `void fA();`) and take no parameters. Functions that have a non-void return type are accepted but the return value is discarded.

Functions that take parameters are ignored with a warning since the parameters would contain garbage values.

Within the same compilation unit, the list of functions in `pragma init` are called in the order specified. Similarly, within the same compilation unit, functions specified in more than one `pragma init` are called in the order in which the pragmas are encountered in the source.

In general, the order of static initialization across files and across libraries is nonstandard and therefore, a non-portable behavior. It is not advisable to build any dependency on this behavior. The order of functions across files is undefined, even when using the **-Wm** option).

When mixing C and C++ files, the relative order of init functions in C files with respect to the static constructors/destructors in C++ files is undefined. The **-qunique** option can interact with `pragma init`.

Note: A C++ invocation, such as **xlC** or the redistributable tools **linkxlC** or **makeC++SharedLib** must be used at link time.

Related information
- "#pragma fini (C only)" on page 325
- "-qunique (C++ only)" on page 288

#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 326

Related information
- "#pragma hashome (C++ only)" on page 326

#pragma isolated_call
See "-qisolated_call" on page 158.

#pragma langlvl (C only)
See "-qlanglvl" on page 165.

#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 348.

#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 315.

Related information
- "-qunroll" on page 290
- "#pragma block_loop" on page 315
- "#pragma unrollandfuse" on page 354

#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. You can also use can the **c++filt** utility provided by the compiler for a side-by-side listing of source names and mangled names; see "Demangling compiled C++ names" in the *XL C/C++ Optimization and Programming Guide* for details. (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, "bar__Fv")
#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 349 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 =
}
```

The result of running the program is:

```
result = 1
```

Related information
- "-qisolated_call" on page 158
- "#pragma reg_killed_by" on page 349
- "Inline assembly statements" in the *XL C/C++ Language Reference*

#pragma namemangling (C++ only)

See "-qnamemangling (C++ only)" on page 198.

#pragma namemanglingrule (C++ only)
Category

Portability and migration

Purpose

Provides fined-grained control over the name mangling scheme in effect for selected portions of source code, specifically with respect to the mangling of cv-qualifiers in function parameters.

When a function name is mangled, repeated function arguments of the same type are encoded according to the following compression scheme:

```
parameter → T param number [_]    #single repeat of a previous parameter
          → N repetition digit param number [_]    #2 to 9 repetitions
```

where:

param number
    Indicates the number of the previous parameter which is repeated. It is followed by an underscore (_) if *param number* contains multiple digits.

repetition digit
    Must be greater than 1 and less than 10. If an argument is repeated more than 9 times, this rule is applied multiple times. For example, a sequence of 38 parameters that are the same as parameter 1 mangles to N91N91N91N91N21.

The **#pragma namemanglingrule** directive allows you to control whether top-level cv-qualifiers are mangled in function parameters or whether intermediate-level cv-qualifiers are to be considered when the compiler compares repeated function parameters for equivalence.

Syntax
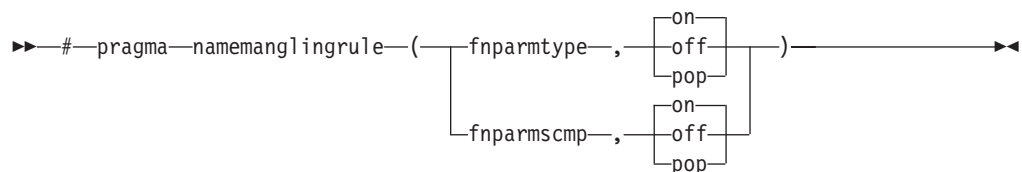
Defaults

- **fnparmtype, on** when **-qnamemangling=ansi|v6** or higher or **#pragma namemangling (ansi|v6)** or higher is in effect. Otherwise, the default is **fnparmtype, off**.
- **fnparmscmp, on** when **-qnamemangling=ansi|v8** or higher or **#pragma namemangling (ansi|v8)** or higher is in effect. Otherwise, the default is **fnparmscmp, off**.

Parameters

fnparmtype, on
    Top-level cv-qualifiers are not encoded in the mangled name of a function parameter. Also, top-level cv-qualifiers are ignored when repeated function parameters are compared for equivalence; function parameters that differ only by the use of a top-level cv-qualifier are considered equivalent and are mangled according to the compressed encoding scheme. This setting is compatible with VisualAge C++ V6.0 and higher.

fnparmtype, off
    Top-level cv-qualifiers are encoded in the mangled name of a function parameter. Also, repeated function parameters that differ by the use of cv-qualifiers are not considered equivalent and are mangled as separate parameters. This setting is compatible with VisualAge C++ V5.0 and earlier.

fnparmtype, pop
    Reverts to the previous **fnparmtype** setting in effect. If no previous settings are in effect, the default **fnparmtype** setting is used.

Note: This pragma fixes function signature ambiguities in 32-bit mode but it is not needed in 64-bit mode since those ambiguities do exist in 64-bit mode.

fnparmscmp, on
Intermediate-level cv-qualifiers are considered when repeated function parameters are compared for equivalence; repeated function parameters that differ by the use of intermediate-level cv-qualifiers are mangled as separate parameters. This setting is compatible with XL C++ V8.0 and higher.

fnparmscmp, off
Intermediate-level cv-qualifiers are ignored when repeated function parameters are compared for equivalence; function parameters that differ only by the use of an intermediate-level cv-qualifier are considered equivalent and are mangled according to the compressed encoding scheme, described in the Notes below. This setting is compatible with XL C++ V7.0 and earlier.

fnparmscmp, pop
Reverts to the previous **fnparmscmp** setting in effect. If no previous settings are in effect, the default **fnparmscmp** setting is used.

Usage

#pragma namemanglingrule is allowed in global, class, and function scopes. It has no effect on a block scope function declaration with external linkage.

Different pragma settings can be specified in front of function declarations and definitions. If **#pragma namemanglingrule** settings in subsequent declarations and definitions conflict, the compiler ignores those settings and issues a warning message.

Examples

The following tables show the effects of this pragma applied to different function signatures.

Table 36. Mangling of function parameters with top-level cv-qualifiers

	Mangled name	
Source name	fnparmtype, off	fnparmtype, on
void foo (const int)	foo__FCi	foo__Fi
void foo (int* const)	foo__FCPi	foo__FPi
void foo (int** const)	foo__FCPPi	foo__FPPi
void foo (int, const int)	foo__FiCi	foo__FiT1

Table 37. Mangling of function parameters with intermediate level cv-qualifiers

	Mangled name	
Source name	fnparmscmp, on	fnparmscmp, off
void foo (int**, int* const *)	foo__FPPiPCPi	foo__FPPiT1

Table 38. Mangling of function parameters with top-level and intermediate-level cv-qualifiers

Source name	Mangled name			
	fnparmscmp, on fnparmtype, on	**fnparmscmp, off fnparmtype, on**	**fnparmscmp, on fnparmtype, off**	**fnparmscmp, off fnparmtype, off**
void foo (int** const, int* const *)	foo__FPPiPCPi	foo__FPPiT1	foo__FCPPiPCPi	foo__FPPiT1

Related information

- "-qnamemangling (C++ only)" on page 198

#pragma nosimd

See "-qhot" on page 132.

#pragma novector

See "-qhot" on page 132.

#pragma object_model (C++ only)

See "-qobjmodel (C++ only)" on page 208.

#pragma operator_new (C++ only)

Category

Error checking and debugging

Purpose

Determines whether the new and new[] operators throw an exception if the requested memory cannot be allocated.

This pragma is equivalent to the **-qlanglvl=newexcp** option.

Syntax

```
>>--#--pragma--operator_new--(--+--returnsnull------+--)----------------><
                                 +--throwsexception--+
```

Defaults

returnsnull

Parameters

returnsnull
If the memory requested by the new operator cannot be allocated, the compiler returns 0, the null pointer. Use this option for compatibility with versions of the XL C++ compiler previous to Visual C++ V6.0.

throwsexception
If the memory requested by the new operator cannot be allocated, the compiler throws a standard exception of type std::bad_alloc. Use this option in new applications, for conformance with the C++ standard.

Usage

The pragma can be specified only once in a source file. It must appear before any statements in the source file. This pragma takes precedence over the **-qlanglvl=newexcp** compiler option.

Restrictions

This pragma applies only to versions of the new operator that throw exceptions; it does not apply to the nothrow or empty throw versions of the new operator (for the prototypes of all the new operator versions, see the description of the <new> header in the *Standard C++ Library Reference*). It also does not apply to class-specific new operators, user-defined new operators, and new operators with placement arguments.

Related information

- "Allocation and deallocation functions" in the *XL C/C++ Language Reference*
- "The new operator" in the *XL C/C++ Language Reference*
- "-qlanglvl" on page 165
- The <new> header in the *Standard C++ Library Reference*

#pragma options
Category

Language element control

Purpose

Specifies compiler options in your source program.

Syntax

```
►►──#──pragma──┬─option──┬──┬──▼─── option_keyword ─────────────────────┬──►◄
               └─options─┘  │                              ┌─;─┐         │
                            │                          ┌─,─┤            │
                            └─ option_keyword ─=─▼──▼── value ─┘        
```
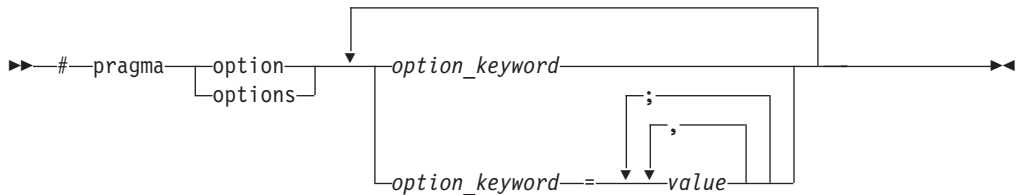
Parameters

The settings in the table below are valid *options* for **#pragma options**. For more information, refer to the pages of the equivalent compiler option.

Valid settings for #pragma options *option_keyword*	Compiler option equivalent
align=*option*	"-qalign" on page 62
[no]attr attr=full	"-qattr" on page 74
chars=*option*	"-qchars" on page 84
[no]check	"-qcheck" on page 85

Valid settings for #pragma options *option_keyword*	Compiler option equivalent
[no]compact	"-qcompact" on page 87
[no]dbcs	"-qmbcs, -qdbcs" on page 193
▶ C [no]dbxextra	"-qdbxextra (C only)" on page 95
[no]digraph	"-qdigraph" on page 97
[no]dollar	"-qdollar" on page 98
enum=*option*	"-qenum" on page 104
[no]extchk	"-qextchk" on page 109
flag=*option*	"-qflag" on page 113
float=[no]*option*	"-qfloat" on page 114
[no]flttrap=*option*	"-qflttrap" on page 119
[no]fullpath	"-qfullpath" on page 124
▶ C++ [no]funcsect	"-qfuncsect" on page 124
halt	"-qhalt" on page 129
[no]idirfirst	"-qidirfirst" on page 136
[no]ignerrno	"-qignerrno" on page 137
ignprag=*option*	"-qignprag" on page 138
[no]info=*option*	"-qinfo" on page 141
initauto=*value*	"-qinitauto" on page 147
[no]inlglue	"-qinlglue" on page 147
isolated_call=*names*	"-qisolated_call" on page 158
▶ C langlvl	"-qlanglvl" on page 165
[no]ldbl128	"-qldbl128, -qlongdouble" on page 179
[no]libansi	"-qlibansi" on page 181
[no]list	"-qlist" on page 182
[no]longlong	"-qlonglong" on page 186
▶ C [no]macpstr	"-qmacpstr" on page 187
[no]maxmem=*number*	"-qmaxmem" on page 192
[no]mbcs	"-qmbcs, -qdbcs" on page 193
[no]optimizeoptimize=*number*	"-O, -qoptimize" on page 204
▶ C++ priority=*number*	"-qpriority (C++ only)" on page 224
proclocal, procimported, procunknown	"-qprocimported, -qproclocal, -qprocunknown" on page 225
▶ C [no]proto	"-qproto (C only)" on page 227
[no]ro	"-qro" on page 235
[no]roconst	"-qroconst" on page 237
[no]showinc	"-qshowinc" on page 244
[no]source	"-qsource" on page 251
spill=*number*	"-qspill" on page 254

Valid settings for #pragma options *option_keyword*	Compiler option equivalent
▶ C [no]srcmsg	"-qsrcmsg (C only)" on page 255
[no]stdinc	"-qstdinc" on page 259
[no]strict	"-qstrict" on page 260
tbtable=*option*	"-qtbtable" on page 270
tune=*option*	"-qtune" on page 283
[no]unrollunroll=*number*	"-qunroll" on page 290
▶ C [no]upconv	"-qupconv (C only)" on page 293
[no]xref	"-qxref" on page 304

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─┘
```
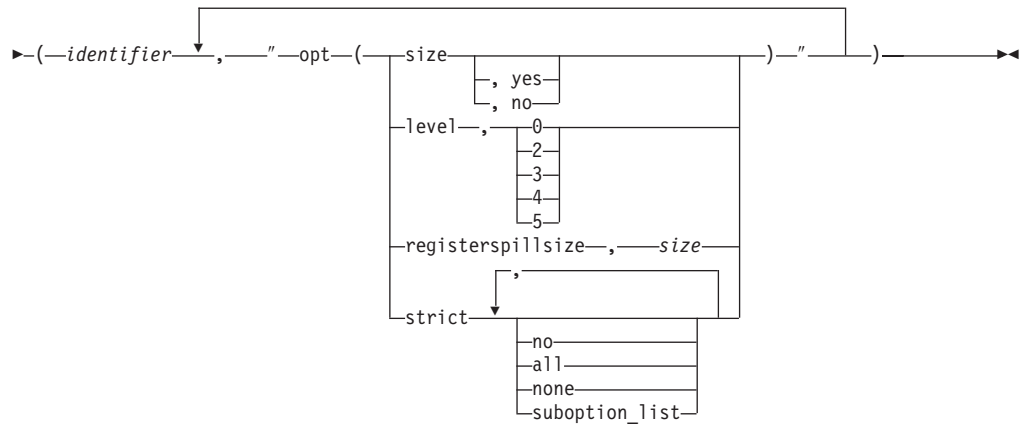
Parameters

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 204
- "-qcompact" on page 87
- "-qspill" on page 254
- "-qstrict" on page 260

#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.

Syntax

Default #pragma pack syntax (-qpack_semantic=ibm in effect)

```
►►─#─pragma─pack─(─────────────)──────────────────────────────◄
                  ├─nopack─┤
                  ├─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.

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

> Removes the previous value added with **#pragma pack**. Specifying **#pragma pack()** with no parameters is equivalent to **pop**.

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 aligns all bit fields in a structure/union on 1-bit boundaries. Example:

```
#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
```
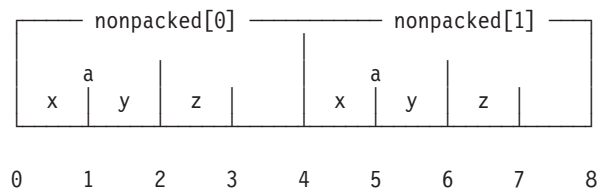
The next example uses **#pragma pack(1)** to set the alignment of unions of type uu
to 1 byte:

```
#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
```
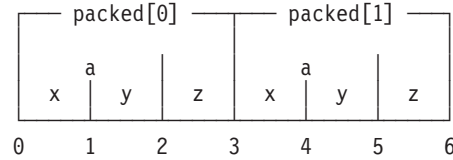
Related information
- "-qalign" on page 62
- "Using alignment modifiers" in the *XL C/C++ Optimization and Programming Guide*

#pragma pass_by_value (C++ only)

See "-qoldpassbyvalue (C++ only)" on page 209.

#pragma priority (C++ only)

See "-qpriority (C++ only)" on page 224.

#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──┴──)──────────────►◄
```
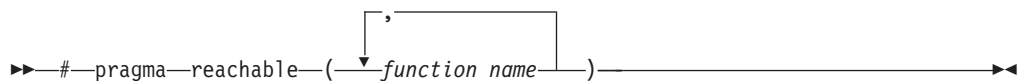
Parameters

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 331

#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─┘
```
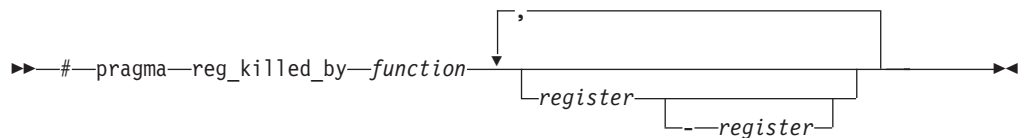
Parameters

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 =
}
```

Related information
* "#pragma mc_func" on page 335

#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

```
►►──#──pragma──report──(──┬──level──,──┬─I─┬──────────────────────┬──)──────────────►◄
                          │            ├─E─┤                      │
                          │            └─W─┘                      │
                          ├──enable──┬──,──message_number─────────┤
                          │  └─disable─┘                          │
                          └──pop──────────────────────────────────┘
```
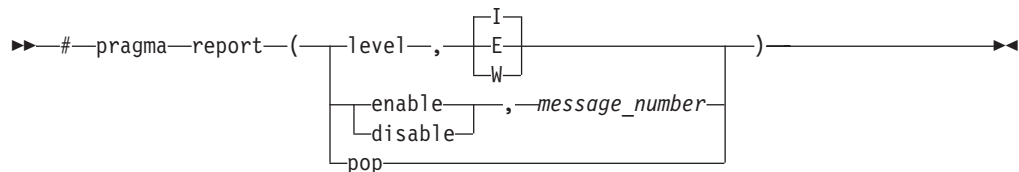
Defaults

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; for 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 113

#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

```
                                      ┌─off─────┐
►►──#──pragma──STDC cx_limited_range──┼─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 290
- "#pragma unrollandfuse" on page 354

#pragma strings

See "-qro" on page 235.

#pragma unroll

See "-qunroll" on page 290.

#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 290
- "#pragma stream_unroll" on page 352

#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. You can also use can the **c++filt** utility provided by the compiler for a side-by-side listing of source names and mangled names; see "Demangling compiled C++ names" in the *XL C/C++ Optimization and Programming Guide* for details. (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**.

#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 foo__Fv
#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 foo__Fv = foo2__Fv
#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 332
- "-qweaksymbol" on page 303
- "-qweakexp" on page 302

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.

You can use IBM SMP or OpenMP directives in C programs, and OpenMP directives in C++ programs. Each has its own usage characteristics.

#pragma ibm critical (C only)
Purpose

The **critical** pragma identifies a critical section of program code that must only be run by one process at a time.

Syntax

```
►►─#─pragma─ibm critical────────────────────────────────────►◄
                          └─(name)─┘
```

where *name* can be used to optionally identify the critical region. Identifiers naming a critical region have external linkage.

Usage

The compiler reports an error if you try to branch into or out of a critical section. Some situations that will cause an error are:

* A critical section that contains the `return` statement.
* A critical section that contains `goto`, `continue`, or `break` statements that transfer program flow outside of the critical section.
* A `goto` statement outside a critical section that transfers program flow to a label defined within a critical section.

#pragma ibm independent_calls (C only)
Description

The **independent_calls** pragma asserts that specified function calls within the chosen loop have no loop-carried dependencies. This information helps the compiler perform dependency analysis.

Syntax

```
                                          ┌─,──────────────┐
►►─#─pragma─ibm independent_calls─────────┴─(identifier)─┴──►◄
```

Where *identifier* is a comma-separated list that represents the name of the functions.

Usage

identifier cannot be the name of a pointer to a function.

If no function identifiers are specified, the compiler assumes that all functions inside the loop are free of carried dependencies.

#pragma ibm independent_loop (C only)
Purpose

The **independent_loop** pragma asserts that iterations of the chosen loop are independent, and that the loop can be parallelized.

Syntax

►►—#—pragma—ibm independent_loop——————————————————————►◄
                                  └─*if exp*─┘

where *exp* represents a scalar expression.

Usage

When the `if` argument is specified, loop iterations are considered independent only as long as *exp* evaluates to TRUE at run time.

This pragma can be combined with the **schedule** pragma to select a specific parallel process scheduling algorithm. For more information, see the "#pragma ibm schedule (C only)" on page 362 description for the **schedule** pragma.

#pragma ibm iterations (C only)
Purpose

The **iterations** pragma specifies the approximate number of loop iterations for the chosen loop.

Syntax

►►—#—pragma—ibm iterations—*(iteration-count)*————————————►◄

where *iteration-count* represents a positive integral constant expression.

Usage

The compiler uses the information in the *iteration-count* variable to determine if it is efficient to parallelize the loop.

#pragma ibm parallel_loop (C only)
Purpose

The **parallel_loop** pragma explicitly instructs the compiler to parallelize the chosen loop.

Syntax

►►—#—pragma—ibm parallel_loop————————————————————————————►◄
                             └─*if exp*─┘ └─*schedule (sched-type)*─┘

where *exp* represents a scalar expression, and *sched-type* represents any scheduling algorithm as valid for the *schedule* directive.

Usage

When the `if` argument is specified, the loop executes in parallel only if *exp* evaluates to TRUE at run time. Otherwise the loop executes sequentially. The loop will also run sequentially if it is in a critical section.

This pragma can be applied to a wide variety of C loops, and the compiler will try to determine if a loop is countable or not.

Program sections using the **parallel_loop** pragma must be able to produce a correct result in both sequential and parallel mode. For example, loop iterations must be independent before the loop can be parallelized. Explicit parallel programming techniques involving condition synchronization are not permitted.

The compiler will not automatically detect reductions on loops marked with this pragma. To properly parallelize loops with reductions, use:
* **parallel for** and specify reductions explicitly, or,
* **#pragma ibm independent_loop**, which will let the compiler discover the reductions.

This pragma can be combined with the **schedule** pragma to select a specific parallel process scheduling algorithm. For more information, see the description for the "#pragma ibm schedule (C only)" on page 362 pragma.

A warning is generated if this pragma is not followed by a countable loop.

#pragma ibm permutation (C only)
Purpose

The **permutation** pragma asserts that on the following loop, different elements of the named arrays are guaranteed to have different values (that is, `a[i] == a[j]` `iff i == j`).

Syntax

```
►►─#─pragma─ibm permutation─┬─(identifier)─┬──────────────────────────►◄
                            └──────,←──────┘
```

where *identifier* represents the name of an array. The *identifier* cannot be a function parameter or the name of a pointer.

Usage

Pragma must appear immediately before the loop or loop block directive to be affected.

This assertion may enable loop transformations if elements are used to index other arrays. This pragma is useful for programs that deal with sparse data structures.

#pragma ibm schedule (C only)
Purpose

The **schedule** pragma specifies the scheduling algorithms used for parallel processing.

Syntax

▶▶──#──pragma──ibm schedule──*(sched-type)*──────────────────────────────────────◀◀

Parameters

sched-type represents one of the following options:

affinity
> Iterations of a loop are initially divided into local partitions of size **ceiling**(*number_of_iterations*/*number_of_threads*). Each local partition is then further subdivided into chunks of size **ceiling**(*number_of_iterations_remaining_in_partition*/2).

> When a thread becomes available, it takes the next chunk from its local partition. If there are no more chunks in the local partition, the thread takes an available chunk from the partition of another thread.

affinity,*n*
> As above, except that each local partition is subdivided into chunks of size *n*. *n* must be an integral assignment expression of value 1 or greater.

dynamic
> Iterations of a loop are divided into chunks of size 1.

> Chunks are 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 that all 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 a chunk size of one is reached. The first chunk is of size **ceiling**(*number_of_iterations*/*number_of_threads*). Remaining chunks are of size **ceiling**(*number_of_iterations_remaining*/ *number_of_threads*).

> 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.

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.
>
> **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*

Usage

Pragma must appear immediately before the loop or loop block directive to be affected.

Scheduling algorithms for parallel processing can be specified using any of the methods shown below. If used, methods higher in the list override entries lower in the list.

- pragma statements
- compiler command line options
- runtime command line options
- runtime default options

Scheduling algorithms can also be specified using the **schedule** argument of the **parallel_loop** and **independent_loop** pragma statements. For example, the following sets of statements are equivalent:

```
#pragma ibm parallel_loop
#pragma ibm schedule (sched_type)
<countable for|while|do loop>
```
and
```
#pragma ibm parallel_loop (sched_type)
<countable for|while|do loop>
```

If different scheduling types are specified for a given loop, the last one specified is applied.

#pragma ibm sequential_loop (C only)
Purpose

The **sequential_loop** pragma explicitly instructs the compiler to execute the chosen loop sequentially.

Syntax

►►──#──pragma──ibm sequential_loop──────────────────────────────────►◄

Usage

Pragma must appear immediately before the loop or loop block directive to be affected.

This pragma disables automatic parallelization of the chosen loop, and is always respected by the compiler.

#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─────┘
```
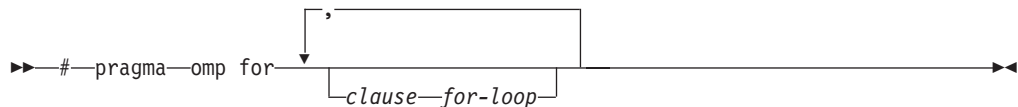
Parameters

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 369 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**,**nonrollandfuse** pragmas cannot be used for any of the loops associated with the **collapse** clause loop nest.

Related reference

#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 for─┴───┴──────────────────────────────►◄
                  └─clause──for-loop─┘
```
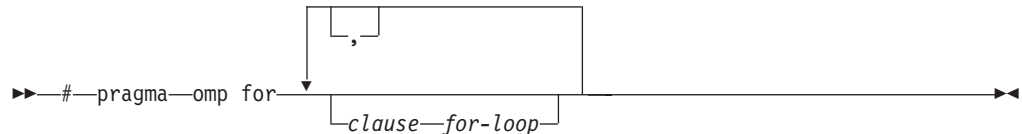
Usage

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─┘
```
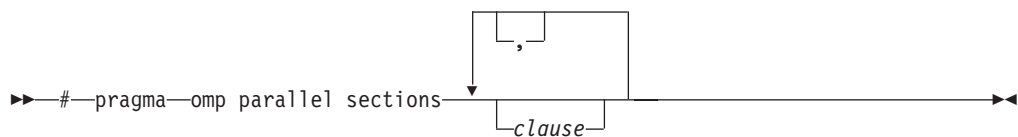
Usage

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
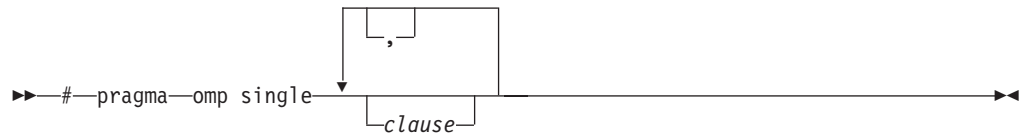
Parameters

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.
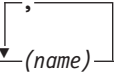
#pragma omp critical
Purpose

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
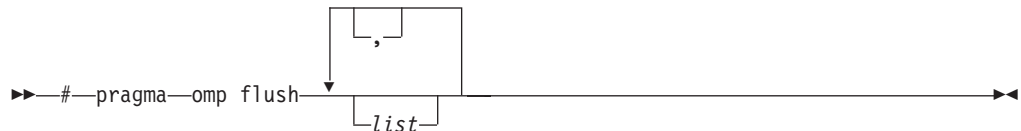
where *list* is a comma-separated list of variables that will be synchronized.

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 376

Chapter 5. 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 381
- "Macros related to compiler features" on page 381

"Examples of predefined macros" on page 389 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 39. 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. `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 39. 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 40. 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++ V10.1, the value of the macro is 1010.
▶ 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++ V10.1, the value of the macro is 1010.
▶ 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++ V10.1, the value of the macro is "10.1.0.0".

Table 40. 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$VVRM$, where: V Represents the version number R Represents the release number M Represents the modification number In XL C/C++ V10.1, the value of the macro is 0x01010.

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 41. 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.
__powerpc, __powerpc__	Indicates that the target architecture is PowerPC.	1	Predefined when the target architecture is PowerPC.
__PPC, __PPC__	Indicates that the target architecture is PowerPC.	1	Predefined when the target architecture is PowerPC.
__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 384
- "Macros related to language levels" on page 385

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 42. 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
__DEBUG_ALLOC__	Indicates that debug versions of the standard memory management functions are being used.	1	-qheapdebug
▶ C++ __EXCEPTIONS	Indicates that C++ exception handling is enabled.	1	-qeh
__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** or**-qlanglvl=strict98 | compat366**
__IBM_DFP__	Indicates support for decimal floating-point types.	1	-qdfp
__IBM_DFP_SW_EMULATION__	Indicates that decimal floating-point computations are implemented through software emulation rather than in hardware instructions.	1	-qfloat=dfpemulate
▶ C _IBMSMP	Indicates that IBM SMP directives are recognized.	1	-qsmp

Table 42. 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_UTF_LITERAL	Indicates support for UTF-16 and UTF-32 string literals.	1	-qutf
▶ 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**
__LONGDOUBLE128	Indicates that the size of a long double type is 128 bits.	1	-qldbl128
▶ C++ __OBJECT_MODEL_CLASSIC__	Indicates that the "classic" object model is in effect.	1	-qobjmodel=classic
▶ C++ __OBJECT_MODEL_IBM__	Indicates that the IBM object is in effect.	1	-qobjmodel=ibm
__OPTIMIZE__	Indicates the level of optimization in effect.	2	-O ǀ -O2
		3	-O3 ǀ -O4 ǀ -O5
__OPTIMIZE_SIZE__	Indicates that optimization for code size is in effect.	1	-O ǀ -O2 ǀ -O3 ǀ -O4 ǀ -O5 and -qcompact
▶ C++ __RTTI_DYNAMIC_CAST__	Indicates that runtime type identification information for the dynamic_cast operator is generated.	1	-qrtti ǀ -qrtti=all ǀ dynamiccast

Table 42. General option-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
▶ C++ __RTTI_TYPE_INFO__	Indicates that runtime type identification information for the typeid operator is generated.	1	-qrtti | -qrtti=all | typeinfo
▶ 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 43. **-qarch**-related macros*

Macro name	Description	Predefined by the following -qarch suboptions
_ARCH_403	Indicates that the application is targeted to run on the PowerPC 403 processor.	403
_ARCH_604	Indicates that the application is targeted to run on the PowerPC 604 processor.	604
_ARCH_COM	Indicates that the application is targeted to run on any PowerPC processor.	Defined for all **-qarch** suboptions except **auto**.
_ARCH_PPC	Indicates that the application is targeted to run on any PowerPC processor.	Defined for all **-qarch** suboptions except **auto**.
_ARCH_PPC64	Indicates that the application is targeted to run on PowerPC processors with 64-bit support.	ppc64 | pwr3 | rs64b | rs64c | ppc64gr | ppc64grsq | ppc64v | pwr4 | pwr5 | pwr5x | pwr6 | pwr6e | ppc970
_ARCH_PPCGR	Indicates that the application is targeted to run on PowerPC processors with graphics support.	ppcgr | 604 | pwr3 | rs64b | rs64c | ppc64gr | ppc64grsq | ppc64v | pwr4 | pwr5 | pwr5x | pwr6 | pwr6e | ppc970
_ARCH_PPC64GR	Indicates that the application is targeted to run on PowerPC processors with 64-bit and graphics support.	pwr3 | rs64b | rs64c | ppc64gr | ppc64v | pwr4 | pwr5 | pwr5x | pwr6 | pwr6e | ppc970
_ARCH_PPC64GRSQ	Indicates that the application is targeted to run on PowerPC processors with 64-bit, graphics, and square root support.	pwr3 | rs64b | rs64c | ppc64grsq | ppc64v | pwr4 | pwr5 | pwr5x | pwr6 | pwr6e | ppc970

Table 43. -qarch-related macros (continued)

Macro name	Description	Predefined by the following -qarch suboptions
_ARCH_PPC64V	Indicates that the application is targeted to run on PowerPC processors with 64-bit and vector processing support.	ppc64v ∣ ppc970 ∣ pwr6 ∣ pwr6e
_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 ∣ ppc970
_ARCH_PWR4	Indicates that the application is targeted to run on POWER4 processors.	pwr4 ∣ pwr5 ∣ pwr5x ∣ pwr6 ∣ pwr6e ∣ ppc970
_ARCH_PWR5	Indicates that the application is targeted to run on POWER5 processors.	pwr5 ∣ pwr5x ∣ pwr6 ∣ pwr6e
_ARCH_PWR5X	Indicates that the application is targeted to run on POWER5+ processors.	pwr5x ∣ pwr6 ∣ pwr6e
_ARCH_PWR6	Indicates that the application is targeted to run on POWER6 processors.	pwr6 ∣ pwr6e
_ARCH_PWR6E	Indicates that the application is targeted to run on POWER6 processors running in POWER6 raw mode.	pwr6e
_ARCH_RS64A	Indicates that the application is targeted to run on the RS64I processor.	rs64a
_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 44. 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
▶ C __C99_COMPLEX	Indicates support for complex data types.	stdc99 ∣ extc99 ∣ extc89 ∣ extended
▶ C++ __C99_COMPLEX_HEADER__	Indicates support for C99-style complex headers.	c99complexheader

Table 44. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
▶ C __C99_CPLUSCMT	Indicates support for C++ style comments	stdc99 | extc99 (also **-qcpluscmt**)
▶ C __C99_COMPOUND_LITERAL	Indicates support for compound literals.	stdc99 | extc99 | extc89 | extended
▶ 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
__C99_EMPTY_MACRO_ARGUMENTS	Indicates support for empty macro arguments.	▶ C stdc99 | extc99 | extc89 | extended ▶ C++ 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 | c99__func__
__C99_HEX_FLOAT_CONST	Indicates support for hexadecimal floating constants.	▶ C stdc99 | extc99 | extc89 | extended ▶ C++ extended | c99hexfloat
▶ C __C99_INLINE	Indicates support for the inline function specifier.	stdc99 | extc99 (also **-qkeyword=inline**)
▶ C __C99_LLONG	Indicates support for C99-style long long data types.	stdc99 | extc99
__C99_MACRO_WITH_VA_ARGS	Indicates support for function-like macros with variable arguments.	▶ C stdc99 | extc99 | extc89 | extended ▶ C++ extended | varargmacros | extended0x
__C99_MAX_LINE_NUMBER	Indicates that the maximum line number is 2147483647.	▶ C stdc99 | extc99 | extc89 | extended ▶ C++ extended0x
▶ C __C99_MIXED_DECL_AND_CODE	Indicates support for mixed declaration and code.	stdc99 | extc99 | extc89 | extended
__C99_MIXED_STRING_CONCAT	Indicates support for concatenation of wide string and non-wide string literals.	▶ C stdc99 | extc99 | extc89 | extended ▶ C++ extended0x
▶ C __C99_NON_LVALUE_ARRAY_SUB	Indicates support for non-lvalue subscripts for arrays.	stdc99 | extc99 | extc89 | extended

Table 44. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
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 (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	c99vla
__DIGRAPHS__	Indicates support for digraphs.	**C** stdc99	extc99	extc89	extended (also **-qdigraph**) **C++** extended (also **-qdigraph**)
C __EXTENDED__	Indicates that language extensions are supported.	extended
__IBM__ALIGN	Indicates support for the `__align` specifier.	**C++** Always defined except when **-qnokeyword=__alignof** is specified
__IBM_ALIGNOF__	Indicates support for the `__alignof__` operator.	**C** extc99	extc89	extended **C++** extended

Table 44. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
__IBM_ATTRIBUTES	Indicates support for type, variable, and function attributes.	`C` extc99 | extc89 | extended `C++` extended
__IBM_COMPUTED_GOTO	Indicates support for computed `goto` statements.	`C` extc99 | extc89 | extended `C++` extended | gnu_computedgoto
__IBM_EXTENSION_KEYWORD	Indicates support for the __extension__ keyword.	`C` extc99 | extc89 | extended `C++` extended
__IBM_GCC__INLINE__	Indicates support for the GCC __inline__ specifier.	`C` extc99 | extc89 | extended `C++` 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 | gnu_labelvalue
__IBM_LOCAL_LABEL	Indicates support for local labels.	`C` extc99 | extc89 | extended `C++` extended | gnu_locallabel
__IBM_MACRO_WITH_VA_ARGS	Indicates support for variadic macro extensions.	`C` extc99 | extc89 | extended `C++` extended | gnu_varargmacros
`C` _IBM_NESTED_FUNCTION	Indicates support for nested functions.	extc99 | extc89 | extended

Table 44. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
`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.
`C++` __IBM_REGISTER_VARIABLES	Indicates support for variables in specified registers.	Always defined.
__IBM__TYPEOF__	Indicates support for the `__typeof__` or `typeof` keyword.	`C` always defined `C++` extended (Also **-qkeyword=typeof**)
`C++` __IBMCPP_TR1__	Indicates support for unordered associative containers (TR1 library extensions) in namespace `std::tr1`.	extended
_LONG_LONG	Indicates support for IBM `long long` data types.	`C` extended | extc89 (also -qlonglong) `C++` extended (also **-qlonglong**)
`C` __SAA__	Indicates that only language constructs that support the most recent level of SAA C standards are allowed.	saa
`C` __SAA_L2__	Indicates that only language constructs that conform to SAA Level 2 C standards are allowed.	saal2

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 6. 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 PowerPC architectures have special instructions that enable the development of highly optimized applications. Access to some PowerPC 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 PowerPC 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 compiler supports all vector processing functions defined by the AltiVec specification. For detailed descriptions of all of these built-in functions, see the *AltiVec Technology Programming Interface Manual*, available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf.

The following sections describe the available built-in functions for the AIX platform.
- "Fixed-point built-in functions"
- "Binary floating-point built-in functions" on page 397
- "Decimal floating-point built-in functions" on page 407
- "Synchronization and atomic built-in functions" on page 425
- "Cache-related built-in functions" on page 432
- "Block-related built-in functions" on page 437
- "Miscellaneous built-in functions" on page 437
- "Built-in functions for parallel processing" on page 441

Fixed-point built-in functions

Fixed-point built-in functions are grouped into the following categories:
- "Absolute value functions" on page 392
- "Assert functions" on page 392
- "Count zero functions" on page 392
- "Load functions" on page 393
- "Multiply functions" on page 393
- "Population count functions" on page 394
- "Rotate functions" on page 394
- "Store functions" on page 396
- "Trap functions" on page 396

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);

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);

Load functions

__load2r, __load4r
Purpose

Load Halfword Byte Reversed, Load Word Byte Reversed

Prototype

> unsigned short __load2r (unsigned short*);

> unsigned int __load4r (unsigned int*);

Multiply functions

__imul_dbl
Purpose

Computes the product of two `long` integers and stores the result in a pointer.

Prototype

> void __imul_dbl (long, long, long*);

__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 Register

Prototype

void __store2r (unsigned short, unsigned short*);

void __store4r (unsigned int, unsigned int*);

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

Binary floating-point built-in functions are grouped into the following categories:

For decimal floating-point built-in functions, see Decimal floating-point built-in functions.

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);

Add functions

__fadd, __fadds
Purpose

Floating Add, Floating Add Single

Adds two arguments and returns the result.

Prototype

> double __fadd (double, double);

> float __fadds (float, 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);

__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);

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

__dfp_set_rounding_mode
Purpose

Set Rounding Mode

Sets the current decimal rounding mode.

Prototype

void __dfp_set_rounding_mode (unsigned long *rounding_mode*);

Parameters

rounding_mode
   One of the compile-time constant values (0 to 7) or macros listed in Table 46 on page 418.

Usage

If you change the rounding mode within a function, you must restore the rounding mode before the function returns.

__dfp_get_rounding_mode
Purpose

Get Rounding Mode

Gets the current decimal rounding mode.

Prototype

unsigned long __dfp_get_rounding_mode (void);

Return value

The current rounding mode as one of the values (0 to 7) listed in Table 46 on page 418.

Multiply functions

__fmul, __fmuls
Purpose

Floating Multiply, Floating Multiply Single

Multiplies two arguments and returns the result.

Prototype

double __fmul (double, double);

float __fmuls (float, float);

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 405.

__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

__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*);

Decimal floating-point built-in functions

Decimal floating-point (DFP) built-in functions are grouped into the following categories:
- "Absolute value functions"
- "Coefficient functions" on page 408
- "Comparison functions" on page 409
- "Conversion functions" on page 410
- "Exponent functions" on page 414
- "NaN functions" on page 415
- "Register transfer functions" on page 416
- "Rounding functions" on page 417
- "Test functions" on page 420

For binary floating-point built-in functions, see Binary floating-point built-in functions

When **-qarch** is set to pwr6 or pwr6e to target POWER6 processors, **-qfloat=nodfpemulate** becomes the default. This means that DFP hardware instructions are generated. Lower-performance software emulation code is generated only when:
- **-qarch** is not set to pwr6 or pwr6e
- **-qarch** is set to pwr6 or pwr6e and **-qfloat=dfpemulate** is enabled

Note: ▶ C++   In the prototypes given in the following sections, the C keyword _Bool is used by convention to represent a Boolean type.

Absolute value functions

Absolute value functions determine the sign of the returned value.

__d64_abs, __d128_abs
Purpose

Absolute Value

Returns the absolute value of the parameter.

Prototype

_Decimal64 __64_abs (_Decimal64);

_Decimal128 __d128_abs (_Decimal128);

__d64_nabs, __d128_nabs
Purpose

Negative Absolute Value

Returns the negative absolute value of the parameter.

Prototype

_Decimal64 __d64_nabs (_Decimal64);

_Decimal128 __d128_nabs (_Decimal128);

__d64_copysign, __d128_copysign
Purpose

Copysign

Returns the absolute value of the first parameter, with the sign of the second parameter.

Prototype

_Decimal64 __d64_copysign (_Decimal64, _Decimal64);

_Decimal128 __d128_copysign (_Decimal128, _Decimal128);

Coefficient functions

Coefficient functions manipulate the fraction without affecting the exponent and sign, to support decimal-floating point conversion library functions.

__d64_shift_left, __d128_shift_left
Purpose

Shift Coefficient Left.

Shifts the coefficient of the parameter left.

Prototype

_Decimal64 __d64_shift_left (_Decimal64, unsigned long *digits*);

_Decimal128 __d128_shift_left (_Decimal128, unsigned long *digits*);

Parameters

digits
> The number of digits to be shifted left. The shift count must be in the range 0 to 63; otherwise the result is undefined.

Return value

The sign and exponent are unchanged. The digits are shifted left.

__d64_shift_right, __d128_shift_right
Purpose

Shift Coefficient Right.

Shifts the coefficient of the parameter right.

Prototype

_Decimal64 __d64_shift_right (_Decimal64, unsigned long *digits*);

_Decimal128 __d128_shift_right (_Decimal128, unsigned long *digits*);

Parameters

digits
> The number of digits to be shifted right. The shift count must be in the range 0 to 63; otherwise the result is undefined.

Return value

The sign and exponent are unchanged. The digits are shifted right.

Comparison functions

Comparison functions support extended exception handling and exponent comparisons.

__d64_compare_exponents, __d128_compare_exponents
Purpose

Compare Exponents

Compares the exponents of two decimal floating-point values.

Prototype

long __d64_compare_exponents (_Decimal64, _Decimal64);

long __d128_compare_exponents (_Decimal128, _Decimal128);

Return value

Returns the following values:
- Less than 0 if the exponent of the first parameter is less than the exponent of the second parameter.
- 0 if both parameters have the same exponent value or if both are quiet or signaling NaNs (quiet and signaling are considered equal) or both are infinities.
- Greater than 0 if the exponent of the first argument is greater than the exponent of the second argument.
- -2 if one of the two parameters is a quiet or signaling NaN or one of the two parameters is an infinity.

__d64_compare_signaling, __d128_compare_signaling
Purpose

Compare Signaling Exception on NaN

Compares two decimal floating-point values and raises an Invalid Operation exception if either is a quiet or signaling NaN.

Prototype

long __d64_compare_signaling (_Decimal64, _Decimal64);

long __d128_compare_signaling (_Decimal128, _Decimal128);

Return value

Returns the following values:
- Less than 0 if the value of the first parameter is less than the value of the second parameter.
- 0 if both parameters have the same value.
- Greater than 0 if the value of the first parameter is greater than the value of the second parameter.

If either value is a quiet or signalling NaN, an exception is raised. If no exception handler has been enabled to trap the exception, the function returns -2.

Usage

If either value is a NaN, normal comparisons using the relational operators (==, !=, <, <=, > and >=) always return false, which raises an exception for a signaling NaN but not for a quiet NaN. If you want an exception to be raised when either value is a quiet or signaling NaN, you should use the Compare Signaling Exception on NaN built-in functions instead of a relational operator.

Conversion functions

Conversion functions execute decimal floating point conversions. Some override the current rounding mode.

__d64_to_long_long, __d128_to_long_long
Purpose

Convert to Integer

Converts a decimal floating-point value to a 64-bit signed binary integer, using the current rounding mode.

Prototype

long long __d64_to_long_long (_Decimal64);

long long __d128_to_long_long (_Decimal128):

Return value

The input value converted to a long long, using the current rounding mode (not always rounded towards zero as a cast or implicit conversion would be).

__d64_to_long_long_rounding, __d128_to_long_long_rounding
Purpose

Convert to Integer

Converts a decimal floating-point value to a 64-bit signed binary integer, using a specified rounding mode.

Prototype

long long __d64_to_long_long_rounding (_Decimal64, long *rounding_mode*);

long long __d128_to_long_long_rounding (_Decimal128, long *rounding_mode*);

Parameters

mode
   One of the compile time constant values or macros defined in Table 46 on page 418.

Return value

The input value converted to a long long, using the specified rounding mode (not always rounded towards zero as a cast or implicit conversion would be).

Usage

These functions temporarily override the rounding mode in effect for the current operation.

__d64_to_signed_BCD
Purpose

Convert to Signed Binary-Coded Decimal

Converts the lower digits of a 64-bit decimal floating-point value to a Signed Packed Format (packed decimal).

Prototype

unsigned long long __d64_to_signed_BCD (_Decimal64, _Bool *value*);

Return value

Produces 15 decimal digits followed by a decimal sign in a 64-bit result. The leftmost digit is ignored.

Positive values are given the sign 0xF if *value* is true and 0xC if *value* is false.

Negative values are given the sign 0xD.

Usage

You can use the `__d64_shift_right` function to access the leftmost digit.

_d128_to_signed_BCD
Purpose

Convert to Signed Binary Coded Decimal.

Converts the lower digits of a 128-bit decimal floating-point value to a Signed Packed Format (packed decimal).

Prototype

void __d128_to_signed_BCD (_Decimal128, _Bool *value*, unsigned long long **upper*, unsigned long long **lower*);

Parameters

upper
    The address of the variable that will hold the upper digits of the result.

lower
    The address of the variable that will hold the lower digits of the result.

Return value

Produces 31 decimal digits followed by a decimal sign in a 128-bit result. Digits to the left are ignored. The higher 16 digits are stored in the parameter *upper*. The lower 15 digits plus the sign are stored in the parameter *lower*.

Positive values are given the sign 0xF if *value* is true and 0xC if *value* is false.

Negative values are given the sign 0xD.

Usage

You can use the `__d128_shift_right` function to access the digits to the left.

__d64_to_unsigned_BCD
Purpose

Convert to Unsigned Binary Coded Decimal.

Converts the lower digits of a 64-bit decimal floating-point value to an Unsigned Packed Format.

Prototype

unsigned long long __d64_to_unsigned_BCD (_Decimal64);

Return value

Returns 16 decimal digits with no sign in a 64-bit result.

Usage

You can use the `__d64_shift_right` function to access the digits to the left.

__d128_to_unsigned_BCD
Purpose

Convert to Unsigned Binary Coded Decimal.

Converts the lower digits of a 128-bit decimal floating-point value to an Unsigned Packed Format.

Prototype

void __d128_to_unsigned_BCD (_Decimal128, unsigned long long *upper*, unsigned long long *lower*);

Parameters

upper
    The address of the variable that will hold the upper digits of the result.

lower
    The address of the variable that will hold the lower digits of the result.

Return value

Produces 32 decimal digits with no sign in a 128-bit result. Digits to the left are ignored. The higher 16 digits are stored in the parameter *upper*. The lower 16 digits are stored in the parameter *lower*.

Usage

You can use the `__d128_shift_right` function to access the digits to the left.

__signed_BCD_to_d64
Purpose

Convert from Signed Binary Coded Decimal.

Converts a 64-bit Signed Packed Format (packed decimal - 15 decimal digits followed by a decimal sign) to a 64-bit decimal floating-point value.

Prototype

_Decimal64 __signed_BCD_to_d64 (unsigned long long);

Parameters

The signs 0xA, 0xC, 0xE, and 0xF are treated as positive. The signs 0xB and 0xD are treated as negative.

__signed_BCD_to_d128
Purpose

Convert from Signed Binary Coded Decimal.

Converts a 128-bit Signed Packed Format (packed decimal - 31 decimal digits followed by a decimal sign) to a 128-bit decimal floating-point value.

Prototype

_Decimal128 __signed_BCD_to_d128 (unsigned long long *upper*, unsigned long long *lower*);

Parameters

upper
    The upper 16 digits of the input value.

lower
    The lower 15 digits plus the sign of the input value.

Parameters

The signs 0xA, 0xC, 0xE, and 0xF are treated as positive. The signs 0xB and 0xD are treated as negative.

__unsigned_BCD_to_d64
Purpose

Convert from Unsigned Binary Coded Decimal.

Converts a 64-bit Unsigned Packed Format (16 decimal digits with no sign) to a 64-bit decimal floating-point value.

Prototype

_Decimal64 __unsigned_BCD_to_d64 (unsigned long long);

__unsigned_BCD_to_d128
Purpose

Convert from Unsigned Binary Coded Decimal.

Converts a 128-bit Unsigned Packed Format (32 decimal digits with no sign) to a 128-bit decimal floating-point value.

Prototype

_Decimal128 __unsigned_BCD_to_d128 (unsigned long long *upper*, unsigned long long *lower*);

Parameters

upper
    The upper 16 digits of the input value.

lower
    The lower 16 digits of the input value.

Exponent functions

Exponent functions extract the exponent from a value or insert an exponent into a value, primarily to support decimal-floating point conversion library functions. They use special values to identify or specify the exponent type.

Table 45. Biased exponents macros and values

Macro	Integer value
DFP_BIASED_EXPONENT_FINITE	0
DFP_BIASED_EXPONENT_INFINITY	-1
DFP_BIASED_EXPONENT_QNAN	-2
DFP_BIASED_EXPONENT_SNAN	-3

__d64_biased_exponent, __d128_biased_exponent
Purpose

Extract Biased Exponent

Returns the exponent of a decimal floating-point value as an integer.

Prototype

long __d64_biased_exponent (_Decimal64);

long __d128_biased_exponent (_Decimal128);

Return value

Returns special values for infinity, quiet NaN, and signalling NaN, as listed in Table 45.

For finite values, the result is DFP_BIASED_EXPONENT_FINITE plus the exponent bias (398 for _Decimal64, 6176 for _Decimal128) plus the actual exponent.

__d64_insert_biased_exponent, __d128_insert_biased_exponent
Purpose

Insert Biased Exponent

Replaces the exponent of a decimal floating-point value.

Prototype

_Decimal64 __d64_insert_biased_exponent (_Decimal64, long *exponent*);

_Decimal128 __d128_insert_biased_exponent (_Decimal128, long *exponent*);

Parameters

exponent
    The exponent value to be applied to the first parameter. For infinity, quiet NaN and signalling NaN, use one of the compile-time constant values or macros listed in Table 45.

    For finite values, the result is DFP_BIASED_EXPONENT_FINITE plus the exponent bias (398 for _Decimal64, 6176 for _Decimal128) plus the desired exponent.

NaN functions

NaN functions create quiet or signaling NaNs.

__d32_sNaN, __d64_sNaN, __d128_sNaN
Purpose

Make Signalling NaN

Creates a signalling NaN of the specified precision, with a positive sign and zero payload.

Prototype

_Decimal32 __d32_sNan (void);

_Decimal64 __d64_sNaN (void);

_Decimal128 __d128_sNaN (void);

__d32_qNaN, __d64_qNaN, __d128qNaN
Purpose

Make Quiet NaN

Creates a quiet NaN of the specified precision, with a positive sign and zero payload.

Prototype

_Decimal32 __d32_qNaN (void);

_Decimal64 __d64_qNaN (void);

_Decimal128 __d128_qNaN (void);

Register transfer functions

Register transfer functions transfer data between general purpose registers and floating-point registers. No conversion occurs. Register transfer functions handle integer data in floating-point registers or floating-point data in general purpose registers. These functions use instructions that are available with **-qarch=pwr6** or **-qarch=pwr6e** only, on a POWER6 running in POWER6e (raw) mode.

__gpr_to_d64
Purpose

Transfer from General Purpose Register to Floating Point Register

Transfers a value from a general-purpose registers (64-bit mode) or a general-purpose register pair (32-bit mode).

Prototype

_Decimal64 __gpr_to_d64 (long long);

__gprs_to_d128
Purpose

Transfer from General Purpose Register to Floating Point Register.

Transfers a value from a pair of general-purpose registers (64-bit mode) or four general-purpose registers (32-bit mode).

Prototype

_Decimal128 __gprs_to_d128 (unsigned long long*_upper_, unsigned long long*_lower_);

Parameters

upper
> The address of the variable that will hold the upper 64 bits of the result.

lower
> The address of the variable that will hold the lower 64 bits of the result.

Return value

The higher 64 bits are stored in the parameter _upper_. The lower 64 bits are stored in the parameter _lower_.

__d64_to_gpr
Purpose

Transfer from Floating Point Register to General Purpose Register.

Transfers a value from a floating-point register to a general-purpose register (64-bit mode) or a general-purpose register pair (32-bit mode).

Prototype

long long __d64_to_gpr (_Decimal64);

__d128_to_gprs
Purpose

Transfer from Floating Point Register to General Purpose Register.

Transfers a value from a pair of floating-point registers to a pair of general-purpose registers (64-bit mode) or four general-purpose registers (32-bit mode).

Prototype

void __d128_to_gprs (_Decimal128, unsigned long long*_upper_, unsigned long long*_lower_);

Parameters

upper
> The address of the variable that contains the upper 64 bits of the input value.

lower
> The address of the variable that contains the lower 64 bits of the input value.

Rounding functions

Rounding functions perform operations such as rounding and truncation of floating-point values.

Table 46. Rounding mode macros and values

Macro	Integer value
DFP_ROUND_TO_NEAREST_WITH_TIES_TO_EVEN	0
DFP_ROUND_TOWARD_ZERO	1
DFP_ROUND_TOWARD_POSITIVE_INFINITY	2
DFP_ROUND_TOWARD_NEGATIVE_INFINITY	3
DFP_ROUND_TO_NEAREST_WITH_TIES_AWAY_FROM_ZERO	4
DFP_ROUND_TO_NEAREST_WITH_TIES_TOWARD_ZERO	5
DFP_ROUND_AWAY_FROM_ZERO	6
DFP_ROUND_TO_PREPARE_FOR_SHORTER_PRECISION	7
DFP_ROUND_USING_CURRENT_MODE[1]	8

Note:

1. This value is valid only for functions that override the current rounding mode; it is not valid for `__dfp_set_rounding_mode` and can not be returned by `__dfp_get_rounding_mode`.

__d64_integral, __d128_integral
Purpose

Round to Integral

Rounds a decimal floating-point value to an integer, allowing an Inexact exception to be raised.

Prototype

_Decimal64 __d64_integral (_Decimal64);

_Decimal128 __d128_integral (_Decimal128);

Return value

The integer is returned in decimal floating-point format, rounded using the current rounding mode. Digits after the decimal point are discarded.

__d64_integral_no_inexact, __d128_integral_no_inexact
Purpose

Round to Integral

Rounds a decimal floating-point value to an integer, suppressing any Inexact exception from being raised.

Prototype

_Decimal64 __d64_integral_no_inexact (_Decimal64);

_Decimal128 __d128_integral_no_inexact (_Decimal128);

Return value

The integer is returned in decimal floating-point format, rounded using the current rounding mode. Digits after the decimal point are discarded.

__d64_quantize, __d128_quantize
Purpose

Quantize

Returns the arithmetic value of the first parameter, with the exponent adjusted to match the second parameter, using a specified rounding mode.

Prototype

_Decimal64 __d64_quantize (_Decimal64, _Decimal64, long *rounding_mode*);

_Decimal128 __d128_quantize (_Decimal128, _Decimal128, long *rounding_mode*);

Parameters

rounding_mode
> One of the compile-time constant values or macros defined in Table 46 on page 418.

Usage

These functions temporarily override the rounding mode in effect for the current operation.

__d64_reround, __d128_reround
Purpose

Reround

Complete rounding of a partially rounded value, avoiding double rounding which causes errors.

Prototype

_Decimal64 __d64_reround (_Decimal64, unsigned long *number_of_digits*, unsigned long *rounding_mode*);

_Decimal128 __d128_reround (_Decimal128, unsigned long *number_of_digits*, unsigned long *rounding_mode*);

Parameters

number_of_digits
> The number of digits to round to, from 1 to 15 for __d64_reround and from 1 to 33 for __d128_reround.

rounding_mode
> One of the compile-time constant values or macros defined in Table 46 on page 418.

Usage

These functions temporarily override the rounding mode in effect for the current operation. The value to be rerounded should have been previously rounded using mode DFP_ROUND_TO_PREPARE_FOR_SHORTER_PRECISION or 7 to ensure correct rounding.

Test functions

Test functions allow extended exception handling of invalid results or categorization of input values, primarily to support math library functions.

Those functions that begin with __d64_is or __d128_is will not raise an exception, even for signaling NaNs.

Table 47. Test data class mask macros and values

Macro	Integer value
DFP_PPC_DATA_CLASS_ZERO	0x20
DFP_PPC_DATA_CLASS_SUBNORMAL	0x10
DFP_PPC_DATA_CLASS_NORMAL	0x08
DFP_PPC_DATA_CLASS_INFINITY	0x04
DFP_PPC_DATA_CLASS_QUIET_NAN	0x02
DFP_PPC_DATA_CLASS_SIGNALING_NAN	0x01

Table 48. Test data group mask macros and values

Macro	Integer value
DFP_PPC_DATA_GROUP_SAFE_ZERO	0x20
DFP_PPC_DATA_GROUP_ZERO_WITH_EXTREME_EXPONENT	0x10
DFP_PPC_DATA_GROUP_NONZERO_WITH_EXTREME_EXPONENT	0x08
DFP_PPC_DATA_GROUP_SAFE_NONZERO	0x04
DFP_PPC_DATA_GROUP_NONZERO_LEFTMOST_DIGIT_NONEXTREME_EXPONENT	0x02
DFP_PPC_DATA_GROUP_SPECIAL	0x01

Table 49. Test data class and group result macros and values

Macro	Integer value
DFP_PPC_DATA_POSITIVE_NO_MATCH	0x00
DFP_PPC_DATA_POSITIVE_MATCH	0x02
DFP_PPC_DATA_NEGATIVE_NO_MATCH	0x08
DFP_PPC_DATA_NEGATIVE_MATCH	0x0A

Table 50. Test data class and group result mask macros and values

Macro	Integer value
DFP_PPC_DATA_NEGATIVE_MASK	0x08
DFP_PPC_DATA_MATCH_MASK	0x02

__d64_same_quantum, __d128_same_quantum
Purpose

Same Quantum

Returns true if two values have the same quantum

Prototype

_Bool __d64_same_quantum (_Decimal64, _Decimal64);

_Bool __d128_same_quantum (_Decimal28, _Decimal128);

__d64_issigned, __d128_issigned
Purpose

Is Signed

Returns true if the parameter is negative, negative zero, negative infinity, or negative NaN.

Prototype

_Bool __d64_issigned (_Decimal64);

_Bool __d128_issigned (_Decimal128);

__d64_isnormal, __d128_isnormal
Purpose

Is Normal

Returns true if the parameter is in the normal range (that is, not a subnormal, infinity or NaN) and not zero.

Prototype

_Bool _d64_isnormal (_Decimal64);

_Bool _d128_isnormal (_Decimal128);

__d64_isfinite, __d128_isfinite
Purpose

Is Finite

Returns true if the parameter is not positive or negative infinity and not a quiet or signaling NaN.

Prototype

_Bool __d64_isfinite (_Decimal64);

_Bool __d128_isfinite (_Decimal128);

__d64_iszero, __d128_iszero
Purpose

Is Zero

Returns true if the parameter is positive or negative zero.

Prototype

_Bool __d64_iszero (_Decimal64);

_Bool __d128_iszero (_Decimal128);

__d64_issubnormal, __d128_issubnormal
Purpose

Is Subnormal

Returns true if the parameter is a subnormal.

Prototype

_Bool _d64_issubnormal (_Decimal64);

_Bool _d128_issubnormal (_Decimal128);

__d64_isinf, __d128_isinf
Purpose

Is Infinity

Returns true if the parameter is positive or negative infinity.

Prototype

_Bool __d64_isinf (_Decimal64);

_Bool __d128_isinf (_Decimal128);

__d64_isnan, __d128_isnan
Purpose

Is NaN

Returns true if the parameter is a positive or negative quiet or signaling NaN.

Prototype

_Bool __d64_isnan (_Decimal64);

_Bool __d128_isnan (_Decimal128);

__d64_issignaling, __d128_issignaling
Purpose

Is Signaling NaN

Returns true if the parameter is a positive or negative signaling NaN.

Prototype

_Bool __d64_issignaling (_Decimal64);

_Bool __d128_issignaling (_Decimal128);

__d64_test_data_class, __d128_test_data_class
Purpose

Test Data Class

Reports if a value is a zero, subnormal, normal, infinity, quiet NaN or signaling NaN, and if the value is positive or negative.

Prototype

long __d64_test_data_class (_Decimal64, unsigned long *mask*);

long __d128_test_data_class (_Decimal128, unsigned long *mask*);

Parameters

mask
> One of the values or macros defined in Table 47 on page 420or several ORed together. The parameter must be a compile time constant expression.

Return value

One of the values listed in Table 49 on page 420.

Usage

You can use an appropriate mask to check combinations of values at the same time. Use the masks listed in Table 47 on page 420 to check input values. Use the masks listed in Table 50 on page 420 to check result values.

__d64_test_data_group, __d128_test_data_group
Purpose

Test Data Group

Reports if a value is a safe zero, a zero with an extreme exponent, a subnormal, a safe nonzero, a normal with no leading zero, or an infinity or NaN and if the value is positive or negative. Safe means leading zero digits and a non-extreme exponent. A subnormal can appear as either an extreme nonzero or safe nonzero. The exact meaning of some masks depends on the particular CPU model.

Prototype

long _d64_test_data_group (_Decimal64, unsigned long *mask*);

long _d128_test_data_group (_Decimal128, unsigned long *mask*);

Parameters

mask
    One of the values or macros defined in Table 48 on page 420 or several ORed
    together. The parameter must be a compile time constant expression.

Return value

One of the values listed in Table 49 on page 420.

Usage

You can use an appropriate mask to check combinations of values at the same
time. Use the masks listed in Table 48 on page 420 to check input values. Use the
masks listed in Table 50 on page 420 to check result values.

__d64_test_significance, __d128_test_significance
Purpose

Test Significance

Checks whether a decimal floating-point value has a specified number of digits of
significance.

Prototype

long __d64_test_significance (_Decimal64, unsigned long *digits*);

long __d128_test_significance (_Decimal128, unsigned long *digits*);

Parameters

digits
    The number of digits of significance to be tested for. *digits* must be in the range
    0 to 63; otherwise the result is undefined. If it is 0, all values including zero
    will be considered to have more significant digits, if it is not 0, a zero value
    will be considered to have fewer significant digits.

Return value

Returns the following values:
- Less than 0 if the number of digits of significance of the first parameter is less
  than the second parameter.
- 0 if the number of digits of significance is the same as the second parameter.
- Greater than 0 if the number of digits of significance of the first parameter is
  greater than that of the second parameter or digits is 0.
- -2 if either parameter is a quiet or signaling NaN or positive or negative infinity.

For these functions, the number of significant digits of the value 0 is considered to
be zero.

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 426
- "Compare and swap functions" on page 427
- "Fetch functions" on page 428
- "Load functions" on page 430
- "Store functions" on page 430
- "Synchronization functions" on page 431

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 438 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, __iospace_sync
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);

> void __iospace_sync (void);

__lwsync, __iospace_lwsync
Purpose

Load Word Synchronize

Ensures that all store instructions preceding the call to __lwsync complete before any new 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
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);

Cache-related built-in functions

Cache-related built-in functions are grouped into the following categories:
- "Data cache functions"
- "Prefetch functions" on page 434
- "Protected stream functions" on page 434

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*);

Prefetch 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 a memory location by using an explicit stream.

Prototype

void __prefetch_by_stream (const int, const void*);

Protected stream functions

__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.

Usage

Valid only when **-qarch** is set to target POWER6 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 value of 0 to 15.

Usage

Valid only when **-qarch** is set to target POWER5 or POWER6 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 value of 0 to 15.

Usage

Valid only when **-qarch** is set to target POWER6 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 or POWER6 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 to 15.

Usage

`__protected_stream_set` and `__protected_unlimited_stream_set_go` are valid only when **-qarch** is set to target POWER5 processors.
`_protected_unlimited_stream_set` is valid only when **-qarch** is set to target POWER5 or POWER6 processors

__protected_stream_stop
Purpose

Stops prefetching a protected stream.

Prototype

> void __protected_stream_stop (unsigned int *stream ID*);

Usage

Valid only when **-qarch** is set to target POWER5 or POWER6 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 or POWER6 processors.

Block-related built-in functions

__bcopy
Purpose

Block copy

Prototype

> void __bcopy (char*, char*, size_t);

__bzero
Purpose

Block zero

Prototype

> void __bzero (char*, int);

Miscellaneous built-in functions

Miscellaneous functions are grouped into the following categories:
- "Optimization-related functions" on page 438
- "Move to/from register functions" on page 439
- "Memory-related functions" on page 441

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.

Related information
*   "Register transfer functions" on page 416

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.

Built-in functions for parallel processing

Use these built-in functions to obtain information about the parallel environment:
*   "IBM SMP built-in functions" on page 442
*   "OpenMP built-in functions" on page 442

IBM SMP built-in functions

__parthds (C only)
Purpose

Returns the value of the **parthds** runtime option.

Prototype

int __parthds (void);

Return value

If the **parthds** option is not explicitly set, returns the default value set by the runtime library. If the **-qsmp** compiler option was not specified during program compilation, returns 1 regardless of runtime options selected.

__usrthds (C only)
Purpose

Returns the value of the **usrthds** runtime option.

Prototype

int __usrthds (void);

Return value

If the **usrthds** option is not explicitly setr, or the **-qsmp** compiler option was not specified during program compilation, returns 0 regardless of runtime options selected.

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. *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. *max-active-levels-var* can be 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 types dynamic, and guided, and 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 types dynamic, and guided, and static, 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 reference**

> "omp_get_schedule" on page 443

> **Related information**

> "OMP_SCHEDULE=*algorithm* environment variable" on page 29

omp_get_thread_limit
Purpose

Retrieves the value of the *thread-limit-var* internal control variable. *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_nested_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.

449

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, 2008. All rights reserved.

Trademarks and service marks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A complete and current list of IBM trademarks is available on the Web 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

-qfdpr compiler option 112
-qoptdebug compiler option 211
-qreport compiler option 233
-qsaveopt compiler option 242
-qsmp compiler option 247
-qunique compiler option 288
-qversion compiler option 296

A

alias 59
  -qalias compiler option 59
  pragma disjoint 320
alignment 62
  -qalign compiler option 62
  pragma align 62
  pragma pack 344
alter program semantics 260
appending macro definitions,
  preprocessed output 245
architecture 9, 67
  -q32 compiler option 58
  -q64 compiler option 58
  -qarch compiler option 67
  -qcache compiler option 81
  -qtune compiler option 283
  architecture combination 285
  macros 384
arrays
  padding 132

B

basic example, described xi
built-in functions 391
  block-related 437
  cache-related 432
  fixed-point 391
  floating-point 397, 407
    binary 397
    decimal 407
  for parallel processing 441
  miscellaneous 437
  synchronization and atomic 425

C

cleanpdf command 217
compatibility
  -qoldpassbyvalue compiler
    option 209
  compatibility
    options for compatibility 53
compiler options 5
  architecture-specific 9
  performance optimization 49
  resolving conflicts 8
  specifying compiler options 5

compiler options *(continued)*
    command line 6
    configuration file 7
    source files 8
  summary of command line
    options 39
configuration 32
  custom configuration files 32
  gxlc and gxlc++ options 36
  specifying compiler options 7
configuration file 110
control of implicit timestamps 277
control of transformations 260

D

data types 67
  -qaltivec compiler option 67
debug optimized code 211
Dynamic Probe Class Library
  -qdpcl compiler option 99
dynamic profiling environment
  variable 30

E

environment variable 23
  environment variables 24
  parallel environment variables 30
  scheduling algorithm environment
    variable 29
  XLSMPOPTS environment
    variable 25
environment variables
  runtime
    XLSMPOPTS 25
error checking and debugging 46
  -g compiler option 126
  -qcheck compiler option 85
  -qheapdebug compiler option 131
  -qlinedebug compiler option 181
exception handling 339
  for floating point 119

F

fini 325
floating-point
  exceptions 119

G

GCC options 11
gxlc and gxlc++ utilities 11

H

high order transformation 132

I

implicit timestamps, control of 277
init pragma 329
inlining 228
interprocedural analysis (IPA) 149
invocations 1
  compiler or components 1
  preprocessor 12
  selecting 1
  syntax 2

L

language level 339
language standards 165
  -qlanglvl compiler option 165
large pages 178
lib*.a library files 163
lib*.so library files 163
libraries
  libraries
    redistributable 15
  XL C/C++ 15
linker 14
  -G compiler option 127
  invoking 14
linking 14
  -brtl compiler option 78
  -G compiler option 127
  options that control linking 52
  order of linking 14
listing 20, 244
  -qattr compiler option 74
  -qlist compiler option 182
  -qlistopt compiler option 184
  -qsource compiler option 251
  -qxref compiler option 304
  options that control listings and
    messages 48

M

machines, compiling for different
  types 67
macro definitions, preprocessed
  output 245
macros 379
  related to architecture 384
  related to compiler options 381
  related to language features 385
  related to the compiler 380
  related to the platform 381
maf suboption of -qfloat 263
mergepdf 217

N

name mangling
  pragma 336

IBM ®

Program Number: 5724-U81

Printed in USA