IBM XL C/C++ Advanced Edition for Linux

Compiler Reference

Version 7.0

SC09-7942-01

IBM XL C/C++ Advanced Edition for Linux

Compiler Reference

Version 7.0

SC09-7942-01

Before using this information and the product it supports, be sure to read the information in “Notices” on page 383.

March 2005 - Second edition

This edition applies to Version 7 Release 0 Modification 1 of IBM XL C/C++ Advanced Edition for Linux (Program number 5724-K77) and to all subsequent releases and modifications until otherwise indicated in new editions.

IBM welcomes your comments. You can send them by Internet: compinfo@ca.ibm.com

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

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

Contents

How to read syntax diagrams

This section describes how to read syntax diagrams. It defines syntax diagram symbols, items that may be contained within the diagrams (keywords, variables, delimiters, operators, fragment references, operands) and provides syntax examples that contain these items.

Syntax diagrams pictorially display the order and parts (options and arguments) that comprise a command statement. They are read from left to right and from top to bottom, following the main path of the horizontal line.

Symbols

The following symbols may be displayed in syntax diagrams:

Symbol	Definition
►►───	Indicates the beginning of the syntax diagram.
───►	Indicates that the syntax diagram is continued to the next line.
►───	Indicates that the syntax is continued from the previous line.
───►◄	Indicates the end of the syntax diagram.

Syntax items

Syntax diagrams contain many different items. Syntax items include:

- Keywords - a command name or any other literal information.
- Variables - variables are italicized, appear in lowercase and represent the name of values you can supply.
- Delimiters - delimiters indicate the start or end of keywords, variables, or operators. For example, a left parenthesis is a delimiter.
- Operators - operators include add (+), subtract (-), multiply (*), divide (/), equal (=), and other mathematical operations that may need to be performed.
- Fragment references - a part of a syntax diagram, separated from the diagram to show greater detail.
- Separators - a separator separates keywords, variables or operators. For example, a comma (,) is a separator.

Keywords, variables, and operators may be displayed as required, optional, or default. Fragments, separators, and delimiters may be displayed as required or optional.

Item type	Definition
Required	Required items are displayed on the main path of the horizontal line.
Optional	Optional items are displayed below the main path of the horizontal line.
Default	Default items are displayed above the main path of the horizontal line.

Syntax examples

The following table provides syntax examples.

Table 1. Syntax examples

<table>
<tr><th>Item</th><th>Syntax example</th></tr>
<tr><td>Required item.

Required items appear on the main path of the horizontal line. You must specify these items.</td><td><pre>
►►──KEYWORD──required_item──────────────────►◄
</pre></td></tr>
<tr><td>Required choice.

A required choice (two or more items) appears in a vertical stack on the main path of the horizontal line. You must choose one of the items in the stack.</td><td><pre>
►►──KEYWORD──┬─required_choice1─┬───────────►◄
             └─required_choice2─┘
</pre></td></tr>
<tr><td>Optional item.

Optional items appear below the main path of the horizontal line.</td><td><pre>
►►──KEYWORD──┬───────────────┬──────────────►◄
             └─optional_item─┘
</pre></td></tr>
<tr><td>Optional choice.

A optional choice (two or more items) appear in a vertical stack below the main path of the horizontal line. You may choose one of the items in the stack.</td><td><pre>
►►──KEYWORD──┬──────────────────┬───────────►◄
             ├─optional_choice1─┤
             └─optional_choice2─┘
</pre></td></tr>
<tr><td>Default.

Default items appear above the main path of the horizontal line. The remaining items (required or optional) appear on (required) or below (optional) the main path of the horizontal line. The following example displays a default with optional items.</td><td><pre>
             ┌─default_choice1──┐
►►──KEYWORD──┼──────────────────┼───────────►◄
             ├─optional_choice2─┤
             └─optional_choice3─┘
</pre></td></tr>
<tr><td>Variable.

Variables appear in lowercase italics. They represent names or values.</td><td><pre>
►►──KEYWORD──variable───────────────────────►◄
</pre></td></tr>
<tr><td>Repeatable item.

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

An arrow returning to the left above a group of repeatable items indicates that one of the items can be selected, or a single item can be repeated.</td><td><pre>
             ┌─────────────────┐
             ▼   │
►►──KEYWORD────repeatable_item─┴────────────►◄
</pre></td></tr>
</table>

Part 1. Overview

XL C/C++

You can use IBM XL C/C++ Advanced Edition for Linux to compile C and C++ program source files into executable object modules.

Note: Throughout these pages, 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.

For more information about the XL C/C++ compiler, see the following topics in this section:

- “Compiler Modes”
- “Compiler Options” on page 5
- “Types of input files” on page 6
- “Types of Output Files” on page 7
- “Compiler Message and Listing Information” on page 8

Compiler Modes

Different forms of 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. The various compiler invocation commands are:

Compiler Invocations	
Basic	**Special**
`xlC`	`xlC_r`
`xlc++`	`xlc++_r`
`xlc`	`xlc_r`
`cc`	`cc_r`
`c99`	`c99_r`
`c89`	`c89_r`
`xlCcore`	`xlCcore_r`

The basic compiler invocation commands appear as the first entry of each line above. Select a basic invocation using the following criteria:

xlC **xlc++**	Both invoke the compiler so that source files are compiled as C++ language source code. If any of your source files are C++, you must use this invocation to link with the correct runtime libraries. Source files are compiled with **-qalias=ansi** set. Files with **.c** suffixes, assuming you have not used the **-+** compiler option, are compiled as C language source code when **-qlanglvl=extc89** is in effect.
xlc	Invokes the compiler for C source files. The following compiler options are implied with this invocation: • -qlanglvl=extc89 • -qalias=ansi • -qcpluscmt • -qkeyword=inline
cc	Invokes the compiler for C source files. The following compiler options are implied with this invocation: • -qlanglvl=extended • -qnoro • -qnoroconst
c99	Invokes the compiler for C source files, with support for ISO C99 language features. Full ISO C99 conformance requires the presence of C99-compliant header files and runtime libraries. The following options are implied with this invocation: • -qlanglvl=stdc99 • -qalias=ansi • -qstrict_induction • -D_ANSI_C_SOURCE • -D_ISOC99_SOURCE • -D__STRICT_ANSI__
c89	Invokes the compiler for C source files, with support for ISO C89 language features. The following options are implied with this invocation: • -qlanglvl=stdc89 • -qalias=ansi • -qstrict_induction • -qnolonglong • -D_ANSI_C_SOURCE • -D__STRICT_ANSI__ Use this invocation for strict conformance to the ANSI standard (ISO/IEC 9899:1990).
xlCcore **xlc++core**	Both will invoke the compiler as described above for **xlC** and **xlc++**, but the compiler will link only to the core of the run-time library. Use this invocation if you want to link your application to a run-time library other than that supplied with XL C/C++.

IBM XL C/C++ Advanced Edition for Linux provides **_r** variations on the basic compiler invocations, as described below:

_r-suffixed Invocations	All **_r**-suffixed invocations additionally define the macro names **_THREAD_SAFE** and **__VACPP_MULTI__** , and add the **-lpthreads**. The compiler option **-qthreaded** is also added. Use these commands if you want to create Posix threaded applications.

Related Tasks

"Invoke the Compiler" on page 19

Related References

"Compiler Command Line Options" on page 35
"General Purpose Pragmas" on page 259
"Pragmas to Control Parallel Processing" on page 323
"threaded" on page 237

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 three ways:

- on the command line
- in a configuration file (.cfg)
- in your source program

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. IBM® 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 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 in a configuration file will override compiler default settings.

Option conflicts that do not follow this priority sequence are described in "Resolving Conflicting Compiler Options" on page 27.

Related Tasks

"Invoke the Compiler" on page 19
"Specify Compiler Options" on page 21
"Resolving Conflicting Compiler Options" on page 27

Related References

"Compiler Command Line Options" on page 35
"General Purpose Pragmas" on page 259
"Pragmas to Control Parallel Processing" on page 323

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 link editor will not be run and temporary object files will be removed.

Your program can consist of several source 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.

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 by specifying either the **-E** or the **-P** option. If you specify the **-P** option, a preprocessed source file, *file_name*.i, is created and processing ends.

The **-E** option preprocesses the source file, writes to standard output, and halts processing without generating an output file.

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

Accepted input file types

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, for example, mysource.c. To use the C++ compiler, the source file must have a **.C** (uppercase C), **.cc**, **.cp**, **.cpp**, **.cxx**, or **.c++** suffix. To compile other files as C++ source files, use the **-+** compiler option. All files specified with this option with a suffix other than **.a**, **.o**, **.s**, or **.so**, are compiled as C++ source files.
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 nonstripped executable files serve as input to the linkage editor. After compilation, the linkage editor 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**. Assembler files are assembled to create an object file.

Accepted input file types

Shared Library Files	Shared library files must have a **.so** suffix, for example*file_name***.so**.

Types of Output Files

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

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

Object Files
The compiler gives object files a **.o** suffix, for example, *file_name***.o**, unless the **-o***file_name* option is specified giving a different suffix or no suffix at all.

If you specify the **-c** option, an output object file, *file_name***.o**, is produced for each input source file *file_name***.***x*, where *x* is a recognized C or C++ filename extension. The linkage editor is not invoked, and the object files are placed in your current directory. All processing stops at the completion of the compilation.

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

Assembler Files
Assembler files must have a **.s** suffix, for example, *file_name***.s**.

They are created by specifying the **-S** option. Assembler files are assembled to create an object file.

Preprocessed Source Files
Preprocessed source files have a **.i** suffix, for example, *file_name***.i**.

To make a preprocessed source file, specify the **-P** option. The source files are preprocessed but not compiled. You can also redirect the output from the **-E** option to generate a preprocessed file that contains #line directives.

A preprocessed source file, *file_name***.i**, is produced for each source file and has the same file name (with a **.i** extension) as the source file from which it was produced.

Listing Files
Listing files have a .lst suffix, for example, *file_name***.lst**.

Specifying any one of the listing-related options to the invocation command produces a compiler listing (unless you have specified the **-qnoprint** option). The file containing this listing is placed in your current directory and has the same file name (with a **.lst** extension) as the source file from which it was produced.

Shared Library Files: Shared library files have a **.so** suffix, for example, my_shrlib.so.

Target Files: Output files associated with the **-M** or **-qmakedep** options have a **.d** suffix, for example, **conversion.d**.

The file contains targets suitable for inclusion in a description file for the **make** command. A **.d** file is created for every input C or C++ file, and is used by the **make** command to determine if a given input file needs to be recompiled as a result of changes made to another input file. **.d** files are not created for any other files (unless you use the **-+** option so other file suffixes are treated as **.C** files).

Related Concepts

"Types of input files" on page 6

Related References

"c" on page 64
"E" on page 88
"M" on page 168
"makedep" on page 171
"o" on page 183
"P" on page 184
"print" on page 194
"S" on page 208

Related References

"c" on page 64
"E" on page 88
"M" on page 168
"makedep" on page 171
"o" on page 183
"P" on page 184
"print" on page 194
"S" on page 208

Compiler Message and Listing Information

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 the appropriate options have been selected, to the listing file.

Compiler Messages

The compiler issues messages 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 in the stderr file. A reconstructed source line is a preprocessed source line that has all the macros expanded.

If you specify the **-qsource** compiler option, the compiler will place messages in the source listing. For example, if you compile your file using the command line invocation **xlc -qsource filename.c**, then you will find a file called **filename.lst** in your current directory.

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.

Compiler Listings

The listings produced by the compiler are useful for debugging. By specifying appropriate options, you can request information on all aspects of a compilation. The listing consists of a combination of the following sections:

- Header section that lists the compiler name, version, and release, as well as the source file name and the date and time of the compilation
- Source section that 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.
- Options section that lists the options that were in effect during the compilation
- Attribute and cross-reference listing section that provides information about the variables used in the compilation unit
- File table section that shows the file number and file name for each main source file and include file
- Compilation epilogue section that summarizes the diagnostic messages, lists the number of source lines read, and indicates whether the compilation was successful
- Object section that lists the object code

Each section, except the header section, has a section heading that identifies it. The section heading is enclosed by angle brackets.

Related References

"Message Severity Levels and Compiler Response" on page 345
"Compiler Message Format" on page 346
"flag" on page 98
"info" on page 119
"source" on page 217
"srcmsg" on page 220
"w" on page 254

Program Parallelization

The compiler lets you explicitly identify sections of C and C++ program code to be parallelized by using pragma directives compliant to the **OpenMP Application Program Interface** specification.

Parallel regions of program code are executed by multiple threads, possibly running on multiple processors. The number of threads created is determined by the run-time options and calls to library functions. Work is distributed among available threads according to the specified scheduling algorithm. For a complete discussion on how threads are created and utilized, refer to the OpenMP Specification, section 2.3 Parallel Construct.

For more information about parallel programming support offered by the XL C/C++ compiler, see the following topics in this section:
- "OpenMP Directives"
- "Shared and Private Variables in a Parallel Environment" on page 12

For complete information about the OpenMP Specification, see:
- OpenMP Web site (www.openmp.org)
- OpenMP Specification (www.openmp.org/specs)

OpenMP Directives

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

Pragmas fall into four general categories:
1. The first category of pragmas lets you define parallel regions in which work is done by threads in parallel. Most of the OpenMP directives either statically or dynamically bind to an enclosing parallel region.
2. The second category lets you define how work will be distributed or shared across the threads in a parallel region.
3. The third category lets you control synchronization among threads.
4. The fourth category lets you define the scope of data visibility across threads.

Related Concepts

"Shared and Private Variables in a Parallel Environment" on page 12

Related Tasks

"Control Parallel Processing with Pragmas" on page 31

Related References

"Pragmas to Control Parallel Processing" on page 323
"OpenMP Run-time Options for Parallel Processing" on page 352
"Built-in Functions Used for Parallel Processing" on page 354

For complete information about the OpenMP Specification, see:
- OpenMP Web site (www.openmp.org)
- OpenMP Specification (www.openmp.org/specs)

Shared and Private Variables in a Parallel Environment

Variables can have either shared or private context in a parallel environment.

- Variables in shared context are visible to all threads running in associated parallel loops.
- Variables in private context are hidden from other threads. Each thread has its own private copy of the variable, and modifications made by a thread to its copy are not visible to other threads.

The default context of a variable is determined by the following rules:

- Variables with **static** storage duration are shared.
- Dynamically allocated objects are shared.
- Variables with automatic storage duration are private.
- Variables in heap allocated memory are shared. There can be only one shared heap.
- All variables defined outside a parallel construct become shared when the parallel loop is encountered.
- Loop iteration variables are private within their loops. The value of the iteration variable after the loop is the same as if the loop were run sequentially.
- Memory allocated within a parallel loop by the **alloca** function persists only for the duration of one iteration of that loop, and is private for each thread.

The following code segments show examples of these default rules:

```
int E1;                         /* shared static     */

 void main (argvc,...) {        /* argvc is shared   */
   int i;                       /* shared automatic  */

   void *p = malloc(...);       /* memory allocated by malloc   */
                                /* is accessible by all threads */
                                /* and cannot be privatized     */

   #pragma omp parallel firstprivate (p)
   {
     int b;                     /* private automatic  */
     static int s;              /* shared static      */

     #pragma omp for
     for (i =0;...) {
       b = 1;                   /* b is still private here !    */
       foo (i);                 /* i is private here because it */
                                /* is an iteration variable     */
     }
     #pragma omp parallel
     {
       b = 1;                   /* b is shared here because it  */
                                /* is another parallel region   */
     }
   }
 }
```

```
int E2;                         /*shared static */

 void foo (int x) {             /* x is private for the parallel */
                                /* region it was called from     */

   int c;                       /* the same */
 ... }
```

The compiler can privatize some shared variables if it is possible to do so without changing the semantics of the program. For example, if each loop iteration uses a unique value of a shared variable, that variable can be privatized. Privatized shared variables are reported by the **-qinfo=private** option. Use critical sections to synchronize access to all shared variables not listed in this report.

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

Data Scope Attribute Clause	Description
private	The **private** clause declares the variables in the list to be private to each thread in a team.
firstprivate	The **firstprivate** clause provides a superset of the functionality provided by the private clause.
lastprivate	The **lastprivate** clause provides a superset of the functionality provided by the private clause.
shared	The **shared** clause shares variables that appear in the list among all the threads in a team. All threads within a team access the same storage area for shared variables.
reduction	The **reduction** clause performs a reduction on the scalar variables that appear in the list, with a specified operator.
default	The **default** clause allows the user to affect the data scope attributes of variables.

For more information, see OpenMP directive descriptions or the OpenMP C and C++ Application Program Interface specification.

Related Concepts

"OpenMP Directives" on page 11

Related Tasks

"Control Parallel Processing with Pragmas" on page 31

Related References

"Pragmas to Control Parallel Processing" on page 323
"OpenMP Run-time Options for Parallel Processing" on page 352
"Built-in Functions Used for Parallel Processing" on page 354

For complete information about the OpenMP Specification, see:

- OpenMP Web site at www.openmp.org
- OpenMP Specification at www.openmp.org/specs

Part 2. Configuration and Use

Set Up the Compilation Environment

Before you compile your C or C++ programs, you must set up the environment variables and the configuration file for your system.

The **vac_configure** program creates an initial configuration file for the compiler, and sets up links to the GCC libraries. For more information on using **vac_configure**, see the *Installation Guide*.

For information about environment variables used by the XL C/C++ compiler, see the following topics in this section:

- "Set Environment Variables"
- "Set Other Environment Variables" on page 18

Set Environment Variables

The Bourne Again SHell (**bash**) on Linux systems is similar to the Bourne Shell (**bsh**) found on AIX® systems. Use the **bash** interface to set the environment variables required by the XL C/C++ compiler, either through the command line or with a command file script.

Set Environment Variables in bash

The following statements, either typed at the command line or inserted into a command file script, show how you can set environment variables in the Bourne Again SHell. Paths shown assume that you are installing the compiler in the default installation location.

```
LANG=en_US
NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/L/%N
export LANG NLSPATH
```

To set the variables so that all users have access to them, 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. The environment variables are set each time the user logs in.

Set Parallel Processing Run-time Options

The XLSMPOPTS environment variable sets options for programs using loop parallelization. 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
```

Additional environment variables set options for program parallelization using OpenMP-compliant directives.

Related Tasks

"Set Environment Variables"

Related References

"Run-time Options for Parallel Processing" on page 349
"OpenMP Run-time Options for Parallel Processing" on page 352

Set Other Environment Variables

Before using the compiler, ensure the following environment variables are set.

PATH
Specifies the directory search path for the executable files of the compiler. Executables are in /opt/ibmcmp/vacpp/7.0/bin if installed to the default location

MANPATH
Optionally specifies the directory search path for finding man pages. MANPATH must contain /opt/ibmcmp/vacpp/7.0/man/en_US before the default man path.

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

LD_RUN_PATH
Specifies the directory search path for dynamically loaded libraries. Used at run time only.

LANG
Specifies the national language for message and help files.

The **LANG** environment variable can be set to any of the locales provided on the system.

The national language code for United States English is **en_US**. If the appropriate message catalogs have been installed on your system, any other valid national language code can be substituted for **en_US**.

To determine the current setting of the national language on your system, use the both of the following **echo** commands:

```
echo $LANG
echo $NLSPATH
```

NLSPATH
Specifies the path name of the message and help files.

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

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

Note: The **LANG** and **NLSPATH** environment variables are initialized when the operating system is installed, and might differ from the ones you want to use.

Related Tasks

"Set Environment Variables" on page 17

Invoke the Compiler

The IBM XL C/C++ compiler is invoked using the following syntax, where *invocation* can be replaced with any valid XL C/C++ invocation command:

```
                  ┌──────────────────────────────────────────┐
                  ▼                                          │
►►──invocation───────┬─────────────────────┬──┬─────────────┬─┴──────────►◄
                     └─command_line_options─┘  └─input_files─┘
```

The parameters of the compiler invocation command can be the names of input files, compiler options, and linkage-editor options. For C++ compiles, the parameters can also include munch options to pass to the munch utility.

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.

To compile without link-editing, 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***.c** input source file, unless the **-o** option was used to specify a different object filename. The linkage editor is not invoked. You can link-edit the object files later using the same invocation command, specifying the object files without the **-c** option.

Note: By default, the invocation command calls *both* the compiler and the linkage editor. It passes linkage editor options to the linkage editor. Consequently, the invocation commands also accept all linkage editor options.

Invoke the Linkage Editor

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

Input Files
Object files and library files serve as input to the linkage editor. Object files must have a **.o** suffix, for example, **year.o**. Static library file names have a **.a** suffix, for example, **libold.a**. Dynamic library file names have a **.so** suffix, for example, **libold.so**.

Output Files
The linkage editor 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 shared object created will have a **.so** filename extension.

You can invoke the linkage editor explicitly with the **ld** command. However, the compiler invocation commands set several linkage-editor 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-edit your object files.

Note: When link-editing object files, *do not* use the **-e** option of the **ld** command. The default entry point of the executable output is `__start`. Changing this label with the **-e** flag can cause erratic results.

Related Concepts

"Compiler Modes" on page 3

Related Tasks

"Specify Compiler Options" on page 21
"Invoke the Compiler" on page 19

Related References

"Compiler Command Line Options" on page 35
"Message Severity Levels and Compiler Response" on page 345
Appendix C, "Libraries in XL C/C++," on page 375

Specify Compiler Options

You can specify compiler options in any of the following ways:

- On the command line (see page 21)
- In your source program (see page 23)
- In a configuration (.cfg) file (see page 24)

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

Specify 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*.

There are two kinds of command-line options:

- **-q***option_keyword* (compiler-specific)
- Flag options

-q Options

```
>>--q--option_keyword--+------------------------+-----------------------><
                       |      .-:---------.     |
                       |      V           |     |
                       '-=------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 **-qflag** option **-qattr** with suboption *full* will produce an attribute listing of all identifiers in the program.

Flag Options

The compilers available on Linux systems use a number of common conventional flag options. IBM XL C/C++ supports these flags. 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 linkage editor, while **-C** can be used with **-P** or **-E** to specify that user comments should be preserved.

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

Related Concepts

"Compiler Options" on page 5

Related Tasks

"Invoke the Compiler" on page 19
"Specify Compiler Options in Your Program Source Files"
"Specify Compiler Options in a Configuration File" on page 24
"Specify Compiler Options for Architecture-Specific, 32- or 64-bit Compilation" on page 25
"Resolving Conflicting Compiler Options" on page 27

Related References

"Compiler Command Line Options" on page 35

Specify Compiler Options in Your 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. It has the general form given below, where *character_sequence* is a series of characters that give specific compiler instruction and arguments, if any.

```
                 ┌──────────────────┐
                 ▼                  │
▶▶──#──pragma────character_sequence─┴──────────────────────────────────▶◀
```

The *character_sequence* on a pragma is subject to macro substitutions, unless otherwise stated. More than one pragma construct can be specified on a single #pragma directive. The compiler ignores unrecognized pragmas, issuing an informational message indicating this.

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 **#pragma** directives are listed in the detailed descriptions of the options to which they apply. For complete details on the various **#pragma** preprocessor directives, see *General Purpose Pragmas*.

Related Concepts

"Compiler Options" on page 5

Related Tasks

"Invoke the Compiler" on page 19
"Specify Compiler Options on the Command Line" on page 21
"Specify Compiler Options in a Configuration File" on page 24
"Specify Compiler Options for Architecture-Specific, 32- or 64-bit Compilation" on page 25
"Resolving Conflicting Compiler Options" on page 27

Related References

"General Purpose Pragmas" on page 259
"Pragmas to Control Parallel Processing" on page 323

Specify Compiler Options in a Configuration File

The default configuration file (**/etc/opt/ibmcmp/vac/7.0/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, and you can make entries to this file to support specific compilation requirements or to support other C or C++ compilation environments.

Most options specified in the configuration file override the default settings of the option. Similarly, most options specified in the configuration file are in turn overridden by options set in the source file and on the command line. Options that do not follow this scheme are listed in *Resolving Conflicting Compiler Options*.

Tailor a Configuration File

The template for the default configuration file template is installed to **/opt/ibmcmp/vac/7.0/etc/vac.base.cfg**.

Before using the compiler for the first time, you must use the **vac_configure** utility to create your own configuration file based on the template file. By default, the **vac_configure** utility creates the new configuration file at **/opt/ibmcmp/vac/7.0/etc/vac.base.cfg**. You can later modify the newly created configuration file to support your specific compilation requirements or to support other C or C++ compilation environments. See the *Installation Guide* for more information on creating configuration files.

You can modify existing stanzas or add new stanzas to a configuration file. For example, to make **-qnoro** the default for the **xlC** compiler invocation command, add **-qnoro** to the **xlC** stanza in your copied version of the configuration file.

You can link the compiler invocation command to several different names. The name you specify when you invoke the compiler determines which stanza of the configuration file the compiler uses. You can add other stanzas to your copy of the configuration file to customize your own compilation environment. You can use the **-F** option with the compiler invocation command to make links to select additional stanzas or to specify a specific stanza in another configuration file. For example:

```
xlc myfile.c -Fmyconfig:SPECIAL
```

would compile myfile.c using the SPECIAL stanza in a myconfig.cfg configuration file that you had created.

Configuration File Attributes

A configuration file includes several stanzas. The following are just some of the items defined by stanzas in the configuration file:

as	Path name to be used for the assembler. The default is **/usr/bin/as**.
ccomp	C Front end. The default is **/opt/ibmcmp/vac/7.0/exe/xlcentry**.
code	Path name to be used for the code generation phase of the compiler. The default is **/opt/ibmcmp/vac/7.0/bin/xlCcode**.
codeopt	List of options for the code-generation phase of the compiler.

crt Path name of the object file passed as the first parameter to the linkage editor. If you do not specify either the **-p** or the **-pg** option, the **crt** value is used. The default is **/usr/lib/crt1.o**.

csuffix Suffix for source programs. The default is **c** (lowercase c).

dis Path name of the disassembler. The default is **/opt/ibmcmp/vac/7.0/exe/dis**.

gcrt Path name of the object file passed as the first parameter to the linkage editor. If you specify the **-pg** option, the gcrt value is used. The default is **/usr/lib/gcrt1.o**.

ld Path name to be used to link C or C++ programs. The default is **/usr/bin/ld**.

ldopt List of options that are directed to the linkage editor part of the compiler. These override all normal processing by the compiler and are directed to the linkage editor. If the corresponding flag takes a parameter, the string is formatted for the **getopt()** subroutine as a concatenation of flag letters, with a letter followed by a colon (**:**).

libraries2 Library options, separated by commas, that the compiler passes as the last parameters to the linkage editor. **libraries2** specifies the libraries that the linkage editor is to use at link-edit time for both profiling and nonprofiling. The default is empty.

mcrt Path name of the object file passed as the first parameter to the linkage editor if you have specified the **-p** option. The default is **/usr/lib/gcrt1.o**.

options A string of option flags, separated by commas, to be processed by the compiler as if they had been entered on the command line.

osuffix The suffix for object files. The default is **.o**.

use Values for attributes are taken from the named stanza and from the local stanza. For single-valued attributes, values in the **use** stanza apply if no value is provided in the local, or default, stanza. For comma-separated lists, the values from the use stanza are added to the values from the local stanza.

xlc The path name of the **xlc** compiler component. The default is **/opt/ibmcmp/vac/7.0/bin/xlc**.

xlC The path name of the **xlC** compiler component. The default is **/opt/ibmcmp/vacpp/7.0/bin/xlC**.

Related Concepts

"Compiler Options" on page 5

Related Tasks

"Invoke the Compiler" on page 19
"Specify Compiler Options on the Command Line" on page 21
"Specify Compiler Options in Your Program Source Files" on page 23
"Specify Compiler Options for Architecture-Specific, 32- or 64-bit Compilation"
"Resolving Conflicting Compiler Options" on page 27

Related References

"Compiler Command Line Options" on page 35

See also the *Configure the compiler* section in *Getting Started with XL C/C++ Advanced Edition for Linux*.

Specify Compiler Options for Architecture-Specific, 32- or 64-bit Compilation

You can use IBM XL C/C++ compiler options to optimize compiler output for use on specific processor architectures. You can also instruct the compiler to compile in either 32- or 64-bit mode.

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

1. Internal default (32-bit mode)
2. Configuration file settings
3. Command line compiler options (**-q32**, **-q64**, **-qarch**, **-qtune**)
4. Source file statements (**#pragma options tune=***suboption*)

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

- *Compiler mode* is set according to the last-found instance of the **-q32** or **-q64** compiler options.
- *Architecture target* is set according to the last-found instance of the **-qarch** compiler option, provided that the specified **-qarch** setting is compatible with the *compiler mode* setting. If the **-qarch** option is not set, the compiler assumes a **-qarch** setting of **ppc64grsq**.
- 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 assumes a **-qtune** setting of **pwr4**.

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

  Resolution: -q32 or **-q64** setting overrides **-qtune** option; compiler issues a warning message, and sets **-qtune** to the **-qarch** setting's default **-qtune** 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- or 64-bit) is determined by the **-q32**/**-q64** compiler settings.

Related Concepts

"Compiler Options" on page 5

Related Tasks

"Invoke the Compiler" on page 19
"Specify Compiler Options on the Command Line" on page 21

Related References

"Compiler Command Line Options" on page 35
"Resolving Conflicting Compiler Options" on page 27

Resolving Conflicting Compiler Options

In general, if more than one variation of the same option is specified (with the exception of **xref** and **attr**), 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 will override compiler options specified on the command line.
2. Compiler options specified on the command line will override compiler options specified 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 in a configuration file will 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.

Option	Conflicting Options	Resolution
-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
-p | **-pg** | **-qprofile**	**-p** | **-pg** | **-qprofile**	Last option specified
-E	**-P** | **-o** | **-S**	**-E**
-P	**-c** | **-o** | **-S**	**-P**
-#	**-v**	**-#**
-F	**-B** | **-t** | **-W** | **-qpath** | configuration file settings	**-B** | **-t** | **-W** | **-qpath**
-qpath	**-B** | **-t**	**-qpath** overrides **-B** and **-t**
-S	**-c**	**-S**

Related Concepts

"Compiler Options" on page 5

Related Tasks

"Invoke the Compiler" on page 19
"Specify Compiler Options on the Command Line" on page 21

Related References

"Compiler Command Line Options" on page 35

Specify Path Names for Include Files

When you imbed one source file in another using the **#include** preprocessor directive, you must supply the name of the file to be included. You can specify a file name either by using a full path name or by using a relative path name.

- **Use a Full Path Name to Imbed Files**

  The *full path name*, also called the *absolute path name*, is the file's complete name starting from the root directory. These path names start with the **/** (slash) character. The full path name locates the specified file regardless of the directory you are presently in (called your *working* or *current* directory).

  The following example specifies the full path to file *mine.h* in John Doe's subdirectory example_prog:

  ```
  /u/johndoe/example_prog/mine.h
  ```

- **Use a Relative Path Name to Imbed Files**

  The *relative path name* locates a file relative to the directory that holds the current source file or relative to directories defined using the **-I***directory* option.

Directory Search Sequence for Include Files Using Relative Path Names

C and C++ define two versions of the **#include** preprocessor directive. IBM XL C/C++ supports both. With the **#include** directive, the include filename is enclosed between either the **< >** or **" "** delimiter characters.

Your choice of delimiter characters will determine the search path used to locate a given include filename. The compiler will search for that include file in all directories in the search path until the include file is found, as follows:

#include type	Directory Search Order
#include <*file_name*>	1. The compiler first searches for *file_name* in each user directory specified by the **-I***directory* compiler option, in the order that they appear on the command line. 2. For C++ compilations, the compiler then searches the directories specified by the **-qcpp_stdinc** and **-qgcc_cpp_stdinc** compiler options. 3. Finally, the compiler then searches the directories specified by the **-qc_stdinc** and **-qgcc_c_stdinc** compiler options.
#include "*file_name*"	1. The compiler first searches for the include file in the directory where your current source file resides. The current source file is the file that contains the directive **#include** "*file_name*". 2. The compiler then searches for the include file according to the search order described above for **#include** <*file_name*>.

Notes:

1. *file_name* specifies the name of the file to be included, and can include a full or partial directory path to that file if you desire.
   - If you specify a file name by itself, the compiler searches for the file in the directory search list.

- If you specify a file name together with a partial directory path, the compiler appends the partial path to each directory in the search path, and tries to find the file in the completed directory path.
- If you specify a full path name, the two versions of the **#include** directive have the same effect because the location of the file to be included is completely specified.

2. The only difference between the two versions of the **#include** directive is that the **" "** (user include) version first begins a search from the directory where your current source file resides. Typically, standard header files are included using the **< >** (system include) version, and header files that you create are included using the **" "** (user include) version.
3. You can change the search order by specifying the **-qstdinc** and **-qidirfirst** options along with the **-I***directory* option.

   Use the **-qnostdinc** option to search only the directories specified with the **-I***directory* option and the current source file directory, if applicable.

   Use the **-qidirfirst** option with the **#include "***file_name***"** directive to search the directories specified with the **-I***directory* option before searching other directories.

   Use the **-I** option to specify the directory search paths.

Related References

"I" on page 115
"c_stdinc" on page 65
"cpp_stdinc" on page 78
"gcc_c_stdinc" on page 108
"gcc_cpp_stdinc" on page 109
"idirfirst" on page 116
"stdinc" on page 224

Control Parallel Processing with Pragmas

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.

OpenMP Directives

▶ C ▶ C++

Syntax

```
#pragma omp pragma_name_and_args
statement_block
```

Pragma directives generally appear immediately before the section of code to which they apply. The **omp parallel** directive is used to define the region of program code to be parallelized. Other OpenMP directives define visibility of data variables in the defined parallel region and how work within that region is shared and synchronized.

For example, the following example defines a parallel region in which iterations of a **for** loop can run in parallel:

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

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

```
#pragma omp sections
{
  #pragma omp section
     structured_block_1
           ...
  #pragma omp section
     structured_block_2
           ...
      ....
}
```

Related Concepts

"Program Parallelization" on page 11

Related References

"Pragmas to Control Parallel Processing" on page 323
"OpenMP Run-time Options for Parallel Processing" on page 352
"Built-in Functions Used for Parallel Processing" on page 354

For complete information about the OpenMP Specification, see:

- OpenMP Web site
- OpenMP Specification

Part 3. Reference

Compiler Options

This section describes the compiler options available in XL C/C++. Options fall into three general groups, as described in the following topics in this section.

- "Compiler Command Line Options"
- "General Purpose Pragmas" on page 259
- "Pragmas to Control Parallel Processing" on page 323

Compiler Command Line Options

This section lists and describes XL C/C++ command line options.

To get detailed information on any option listed, see the full description page(s) for that option. Those pages describe each of the compiler options, including:

- The command-line syntax of the compiler option. The first line under the **Syntax** heading specifies the command-line or configuration-file method of specification. The second line, if one appears, is the **#pragma options** keyword for use in your source file.
- The default setting of the option if you do not specify the option on the command line, in the configuration file, or in a **#pragma** directive within your program.
- The purpose of the option and additional information about its behavior. Unless specifically noted, all options apply to both C and C++ program compilations.

Related Concepts

"Compiler Options" on page 5

Related Tasks

"Specify Compiler Options on the Command Line" on page 21
"Specify Compiler Options in Your Program Source Files" on page 23
"Specify Compiler Options in a Configuration File" on page 24
"Specify Compiler Options for Architecture-Specific, 32- or 64-bit Compilation" on page 25
"Resolving Conflicting Compiler Options" on page 27

Related References

"General Purpose Pragmas" on page 259
"Pragmas to Control Parallel Processing" on page 323

Summary of Command Line Compiler Options

Option Name	Type	Default	Description
+ (plus sign)	*-flag*	-	C++ Compiles any file, *filename.nnn*, as a C++ language file, where *nnn* is any suffix other than **.o**, **.a**, or **.s**.
# (pound sign)	*-flag*	-	Traces the compilation without doing anything.
32, 64	**-q***opt*	**32**	Selects 32- or 64-bit compiler mode.
abi_version	**-q***opt*	See **abi_version**.	C++ Specifies a C++ ABI version for binary compatibility with different levels of GNU C++.
aggrcopy	**-q***opt*	See **aggrcopy**.	Enables destructive copy operations for structures and unions.
alias	**-q***opt*	See **alias**.	Specifies which type-based aliasing is to be used during optimization.
align	**-q***opt*	**linuxppc**	Specifies what aggregate alignment rules the compiler uses for file compilation.
alloca	**-q***opt*	-	C Substitutes inline code for calls to function **alloca** as if **#pragma alloca** directives are in the source code.
altivec	**-q***opt*	**noaltivec**	Enables compiler support for AltiVec™ data types.
arch	**-q***opt*	**arch=ppcv**	Specifies the architecture on which the executable program will be run.
asm	**-q***opt*	**-qasm=gcc**	Controls the interpretation of and subsequent code generation for **asm** statements.
attr	**-q***opt*	**noattr**	Produces a compiler listing that includes an attribute listing for all identifiers.
B	*-flag*	-	Determines substitute path names for the compiler, assembler, linkage editor, and preprocessor.
bigdata	**-q***opt*	**nobigdata**	In 32-bit mode, allows initialized data to be larger than 16 MB in size.
bitfields	**-q***opt*	**bitfields=signed**	Specifies if bit fields are signed.
C	*-flag*	-	Preserves comments in preprocessed output.
c	*-flag*	-	Instructs the compiler to pass source files to the compiler only.
c_stdinc	**-q***opt*	-	C Changes the standard search location for the C headers.
cache	**-q***opt*	-	Specify a cache configuration for a specific execution machine.
chars	**-q***opt*	**chars=unsigned**	Instructs the compiler to treat all variables of type **char** as either **signed** or **unsigned**.
check	**-q***opt*	**nocheck**	Generates code which performs certain types of run-time checking.
cinc	**-q***opt*	**nocinc**	C++ Instructs the compiler to place an **extern ″C″ { }** wrapper around the contents of an include file.

Option Name	Type	Default	Description
compact	**-q***opt*	**nocompact**	When used with optimization, reduces code size where possible, at the expense of execution speed.
complexgccincl	**-q***opt*	**complexgccincl =/usr/include**	Instructs the compiler to internally wrap **#pragma complexgcc(on)** and **#pragma complexgcc(pop)** directives around include files found in specified directories.
cpluscmt	**-q***opt*	See **cpluscmt**.	C Use this option if you want C++ comments to be recognized in C source files.
cpp_stdinc	**-q***opt*	-	C++ Changes the standard search location for the C++ headers.
crt	**-q***opt*	**crt**	Instructs the linker to use the standard system startup files at link time.
D	*-flag*	-	Defines the identifier *name* as in a **#define** preprocessor directive.
dataimported	**-q***opt*	-	Mark data as imported.
datalocal	**-q***opt*	-	Marks data as local.
mbcs, dbcs	**-q***opt*	**nodbcs**	Use the **-qdbcs** option if your program contains multibyte characters.
dbxextra	**-q***opt*	**nodbxextra**	C Specifies that all **typedef** declarations, **struct**, **union**, and **enum** type definitions are included for debugger processing.
digraph	**-q***opt*	See **digraph**.	Enables the use of digraph character sequences in your program source.
directstorage	**-q***opt*	**nodirectstorage**.	Informs the compiler that write-through enabled or cache-inhibited storage may be referenced.
dollar	**-q***opt*	**nodollar**	Allows the $ symbol to be used in the names of identifiers.
E	*-flag*	-	Instructs the compiler to preprocess the source files.
e	*-flag*	-	Specifies the entry name for the shared object. Equivalent to using **ld -e** *name*. See your system documentation for additional information about **ld** options.
eh	**-q***opt*	**eh**	C++ Controls exception handling.
enablevmx	**-q***opt*	**noenablevmx**	Enables generation of VMX (Vector Multimedia Extension) instructions.
enum	**-q***opt*	See **enum**.	Specifies the amount of storage occupied by the enumerations.
F	*-flag*	-	Names an alternative configuration file for the compiler.
flag	**-q***opt*	**flag=i:i**	Specifies the minimum severity level of diagnostic messages to be reported.
float	**-q***opt*	See **float**.	Specifies various floating point options to speed up or improve the accuracy of floating point operations.
flttrap	**-q***opt*	**noflttrap**	Generates extra instructions to detect and trap floating point exceptions.

Option Name	Type	Default	Description
fullpath	**-q***opt*	**nofullpath**	Specifies what path information is stored for files when you use the **-g** option.
funcsect	**-q***opt*	**nofuncsect**	Place instructions for each function in a separate object file control section or csect.
g	*-flag*	-	Generates debugging information for use by a debugger.
gcc_c_stdinc	**-q***opt*	-	C Changes the standard search location for the gcc headers.
gcc_cpp_stdinc	**-q***opt*	-	C++ Changes the standard search location for the g++ headers.
genproto	**-q***opt*	**nogenproto**	C Produces ANSI prototypes from K&R function definitions.
halt	**-q***opt*	C **halt=s** C++ **halt=e**	Instructs the compiler to stop after the compilation phase when it encounters errors of specified *severity* or greater.
haltonmsg	**-q***opt*	-	C++ Instructs the compiler to stop after the compilation phase when it encounters a specific error message.
hot	**-q***opt*	**nohot**	Instructs the compiler to perform high-order loop analysis and transformations during optimization.
I	*-flag*	-	Specifies an additional search path for **#include** filenames that do not specify an absolute path.
idirfirst	**-q***opt*	**noidirfirst**	Specifies the search order for files included with the **#include "** *file_name***"** directive.
ignerrno	**-q***opt*	**noignerrno**	Allows the compiler to perform optimizations that assume **errno** is not modified by system calls.
ignprag	**-q***opt*	-	Instructs the compiler to ignore certain pragma statements.
info	**-q***opt*	C **noinfo** C++ **info=lan**	Produces informational messages.
initauto	**-q***opt*	**noinitauto**	Initializes automatic storage to a specified two-digit hexadecimal byte value.
inlglue	**-q***opt*	**noinlglue**	Generates fast external linkage by inlining the pointer glue code necessary to make a call to an external function or a call through a function pointer.
inline	**-q***opt*	See **inline**.	Attempts to inline functions instead of generating calls to a function.
ipa	**-q***opt*	See **ipa**.	Turns on or customizes a class of optimizations known as interprocedural analysis (IPA).
isolated_call	**-q***opt*	-	Specifies functions in the source file that have no side effects.
keepparm	-q*opt*	**nokeepparm**	C++ Ensures that function parameters are stored on the stack even if the application is optimized.
keyword	-q*opt*	See **keyword**.	Controls whether a specified string is treated as a keyword or an identifier.

Option Name	Type	Default	Description
L	*-flag*	See **L**.	Searches the specified directory at link time for library files specified by the **-l** option.
l	*-flag*	See **l**.	Searches a specified library for linking.
langlvl	**-q***opt*	See **langlvl**.	Selects the C or C++ language level for compilation.
lib	**-q***opt*	**lib**	Instructs the compiler to use the standard system libraries at link time.
libansi	**-q***opt*	**nolibansi**	Assumes that all functions with the name of an ANSI C library function are in fact the system functions.
linedebug	**-q***opt*	**nolinedebug**	Generates abbreviated line number and source file name information for the debugger.
list	**-q***opt*	**nolist**	Produces a compiler listing that includes an object listing.
listopt	**-q***opt*	**nolistopt**	Produces a compiler listing that displays all options in effect.
longlit	**-q***opt*	**nolonglit**	Makes unsuffixed literals the long type for 64-bit mode.
longlong	**-q***opt*	See **longlong**.	Allows **long long** types in your program.
M	*-flag*	-	Creates an output file that contains targets suitable for inclusion in a description file for the **make** command.
makedep	**-q***opt*	-	Creates an output file that contains targets suitable for inclusion in a description file for the **make** command.
maxerr	**-q***opt*	**nomaxerr**	Instructs the compiler to halt compilation when a specified number of errors of specified or greater severity is reached.
maxmem	**-q***opt*	**maxmem=8192**	Limits the amount of memory used for local tables of specific, memory-intensive optimizations.
mbcs, dbcs	**-q***opt*	**nombcs**	Use the **-qmbcs** option if your program contains multibyte characters.
mkshrobj	**-q***opt*	-	Creates a shared object from generated object files.
minimaltoc	**-q***opt*	**nominimaltoc**	Avoids toc overflow conditions in 64-bit compilations by placing toc entries into a separate data section for each object file.
O, optimize	**-q***opt*, *-flag*	**nooptimize**	Optimizes code at a choice of levels during compilation.
o	*-flag*	-	Specifies an output location for the object, assembler, or executable files created by the compiler.
P	*-flag*	-	Preprocesses the C or C++ source files named in the compiler invocation and creates an output preprocessed source file for each input source file.
p	*-flag*	-	Sets up the object files produced by the compiler for profiling.

Option Name	Type	Default	Description
path	**-q***opt*	-	Constructs alternate program and path names.
pdf1, pdf2	**-q***opt*	**nopdf1**, **nopdf2**	Tunes optimizations through Profile-Directed Feedback.
pg	*-flag*	-	Sets up the object files for profiling.
phsinfo	**-q***opt*	**nophsinfo**	Reports the time taken in each compilation phase.
pic	**-q***opt*	**nopic**	Instructs the compiler to generate Position-Independent Code suitable for use in shared libraries.
prefetch	**-q***opt*	**prefetch**	Enables generation of prefetching instructions in compiled code.
print	**-q***opt*	**print**	**-qnoprint** suppresses listings.
priority	**-q***opt*	-	C++ Specifies the priority level for the initialization of static objects.
proclocal, procimported, procunknown	**-q***opt*	See **proclocal**.	Mark functions as local, imported, or unknown.
proto	**-q***opt*	**noproto**	C Assumes all functions are prototyped.
Q	*-flag*	See **Q**.	Attempts to inline functions.
R	*-flag*	See **R**.	Searches the specified directory at run time for shared libraries.
r	*-flag*	-	Produces a relocatable object.
report	**-q***opt*	**noreport**	Instructs the compiler to produce transformation reports that show how program loops are parallelized and optimized.
ro	**-q***opt*	See **ro**.	Specifies the storage type for string literals.
roconst	**-q***opt*	See **roconst**.	Specifies the storage location for constant values.
rtti	**-q***opt*	**rtti**	C++ Generates run-time type identification (RTTI) information for the typeid operator and the dynamic_cast operator.
S	*-flag*	-	Generates an assembly language file (**.s**) for each source file.
s	*-flag*	-	Strips symbol table.
saveopt	**-q***opt*	**nosaveopt**	Saves the command line compiler options into an object file.
showinc	**-q***opt*	**noshowinc**	Used together with **-qsource** to selectively show user header files (includes using " ") or system header files (includes using < >) in the program source listing.
showpdf	**-q***opt*	**noshowpdf**	Used together with **-qpdf1** to add additional call and block count profiling information to an executable.
smallstack	**-q***opt*	**nosmallstack**	Instructs the compiler to reduce the size of the stack frame.
smp	**-q***opt*	**nosmp**	Enables parallelization of program code.
source	**-q***opt*	**nosource**	Produces a compiler listing and includes source code.

Option Name	Type	Default	Description
sourcetype	**-q***opt*	**sourcetype=default**	Instructs the compiler to treat all source files as if they are the source type specified by this option, regardless of actual source filename suffix.
spill	**-q***opt*	**spill=512**	Specifies the size of the register allocation spill area.
srcmsg	**-q***opt*	**nosrcmsg**	C Adds the corresponding source code lines to the diagnostic messages in the **stderr** file.
staticinline	**-q***opt*	**nostaticinline**	Treats inline functions as being static.
staticlink	**-q***opt*	**nostaticlink**	Controls linking to shared libraries.
statsym	**-q***opt*	**nostatsym**	Adds user-defined, non-external names that have a persistent storage class to the name list.
stdinc	**-q***opt*	**stdinc**	Specifies which files are included with **#include <***file_name***>** and **#include "***file_name***"** directives.
strict	**-q***opt*	See **strict**.	Turns off aggressive optimizations of the **-O3** option that have the potential to alter the semantics of your program.
strict_induction	**-q***opt*	See **strict_induction**.	Disables loop induction variable optimizations that have the potential to alter the semantics of your program.
suppress	**-q***opt*	See **suppress**.	Specifies compiler message numbers to be suppressed.
symtab	**-q***opt*	-	Determines what information appears in the symbol table.
syntaxonly	**-q***opt*	-	C Causes the compiler to perform syntax checking without generating an object file.
t	*-flag*	See **t**.	Adds the prefix specified by the **-B** option to designated programs.
tabsize	**-q***opt*	**tabsize=8**	Changes the length of tabs as perceived by the compiler.
tbtable	**-q***opt*	See **tbtable**.	Sets traceback table characteristics.
tempinc	**-q***opt*	See **tempinc**.	C++ Generates separate include files for template functions and class declarations, and places these files in a directory which can be optionally specified.
templaterecompile	**-q***opt*	See **templaterecompile**.	C++ Helps manage dependencies between compilation units that have been compiled using the **-qtemplateregistry** compiler option.
templateregistry	**-q***opt*	**templateregistry**	C++ Maintains records of all templates as they are encountered in the source and ensures that only one instantiation of each template is made.
tempmax	**-q***opt*	**tempmax=1**	C++ Specifies the maximum number of template include files to be generated by the **tempinc** option for each header file.
threaded	**-q***opt*	See **threaded**.	Indicates that the program will run in a multi-threaded environment.
tls	**-q***opt*	See **tls**.	Marks data as local.

Option Name	Type	Default	Description
tmplparse	**-q***opt*	**tmplparse=no**	C++ Controls whether parsing and semantic checking are applied to template definition implementations.
tocdata	**-q***opt*	**notocdata**.	Specifies the thread-local storage model to be used by the application.
trigraph	**-q***opt*	See **trigraph**.	Enables the use of trigraph character sequences in your program source.
tune	**-q***opt*	See **tune**.	Specifies the architecture for which the executable program is optimized.
U	*-flag*	-	Undefines a specified identifier defined by the compiler or by the **-D** option.
unroll	**-q***opt*	**unroll=auto**	Unrolls inner loops in the program.
unwind	**-q***opt*	**unwind**	Informs the compiler that the application does not rely on any program stack unwinding mechanism.
upconv	**-q***opt*	**noupconv**	C Preserves the **unsigned** specification when performing integral promotions.
utf	**-q***opt*	**noutf**	Enables recognition of UTF literal syntax.
V	*-flag*	-	Instructs the compiler to report information on the progress of the compilation in a command-like format.
v	*-flag*	-	Instructs the compiler to report information on the progress of the compilation.
vftable	**-q***opt*	**vftable**	C++ Controls the generation of virtual function tables.
vrsave	**-q***opt*	**vrsave**	Controls function prolog and epilog code necessary to maintain the VRSAVE register.
W	*-flag*	-	Passes the listed words to a designated compiler program.
w	*-flag*	-	Requests that warning messages be suppressed.
warn64	**-q***opt*	**nowarn64**	Enables checking for possible data conversion problems between 32-bit and 64-bit compiler modes.
xcall	**-q***opt*	**noxcall**	Generates code to treat static routines within a compilation unit as if they were external calls.
xref	**-q***opt*	**noxref**	Produces a compiler listing that includes a cross-reference listing of all identifiers.
y	*-flag*	-	Specifies the compile-time rounding mode of constant floating-point expressions.

+ (plus sign)

C++

Purpose

Compiles any file, *filename.nnn*, as a C++ language file, where *nnn* is any suffix other than **.a**, **.o**, or **.s**.

Syntax

```
►►── -+ ──────────────────────────────────────────────────►◄
```

Notes

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.

Example

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

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

Related References

"Compiler Command Line Options" on page 35
"sourcetype" on page 218

(pound sign)

C C++

Purpose

Traces the compilation without invoking anything. This option previews the compilation steps specified on the command line. When the **xlc++** command is issued with this option, it names the programs within the preprocessor, compiler, and linkage editor that would be invoked, and the options that would be specified to each program. The preprocessor, compiler, and linkage editor are not invoked.

Syntax

```
►►── -# ──────────────────────────────────────────────────────►◄
```

Notes

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. Information is displayed to standard output.

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

Example

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

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

Related References

"Compiler Command Line Options" on page 35
"v" on page 250

32, 64

C C++

Purpose

Selects either 32- or 64-bit compiler mode.

Syntax

```
            ┌─32─┐
►►── -q─────┴─64─┴──────────────────────────────────────────────────►◄
```

Notes

If this option is not explicitly specified on the command line, the compiler will default to 32-bit output mode.

If the compiler is invoked in 64-bit mode, the __64BIT__ preprocessor macro is defined.

Use **-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.

Example

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
```

Important!

- If you mix 32-and 64-bit compilation modes for different source files, your objects will not bind. You must recompile completely to ensure that all objects are in the same mode.
- Your link options must reflect the type of objects you are linking. If you compiled 64-bit objects, you must link these objects using 64-bit mode.

Related references

- See "32- and 64-bit application development" in *Getting Started with XL C/C++*
- “arch” on page 56
- “tune” on page 242

abi_version

C++

Purpose

Specifies the C++ ABI version for binary compatibility with different levels of GNU C++.

Syntax

```
>>── -q──abi_version──=──┬─1─┬──────────────────────────────────────────><
                         └─2─┘
```

where:

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

2 Specifies the same C++ ABI behavior as in GNU C++ 3.4, if this version is supported by the Linux operating system.

Notes

The option **-qabi_version** is provided for compatibility with the GNU C++ option -fabi-version=*n*, which allows the user to specify the version of the C++ abstract binary interface used during compilation. The default setting of **-qabi_version** depends on the compiling machine itself and the level of GNU C++ configured during the installation of XL C++. The default is **-qabi_version=1** if GNU C++ 3.2 or 3.3 is installed on the compiling machine.

Informational messages

The value of **-qabi_version** can be ascertained by compiling with **-qlistopt** in effect.

Related references

• "listopt" on page 164

aggrcopy

C C++

Purpose

Enables destructive copy operations for structures and unions.

Syntax

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

Default Setting

The default setting of this option is **-qaggrcopy=overlap** when compiling with **-qlanglvl=extended** or **-qlanglvl=classic** in effect. Otherwise, the default is **-qaggrcopy=nooverlap**.

Programs that do not comply to the ANSI C standard as it pertains to non-overlap of source and destination assignment may need to be compiled with the **-qaggrcopy=overlap** compiler option.

Notes

If the **-qaggrcopy=nooverlap** compiler option is enabled, the compiler assumes that the source and destination for structure and union assignments do not overlap. This assumption lets the compiler generate faster code.

Example

```
xlc myprogram.c -qaggrcopy=nooverlap
```

Related References

"Compiler Command Line Options" on page 35
"langlvl" on page 143

alias

C C++

Purpose

Instructs the compiler to apply aliasing assertions to your compilation unit. The compiler will take advantage of the aliasing assertions to improve optimizations where possible, unless you specify otherwise.

Syntax

```
                    ┌─:──────────────┐
                    │ ┌─noaddrtaken─┐│
                    │ ├─noallptrs───┤│
                    │ ├─typeptr─────┤│
                    │ ├─ansi────────┤│
                    ▼ │             ││
►►── -q──alias──=───┴─┼─noansi──────┼┴──────────────────────────►◄
                      ├─notypeptr───┤
                      ├─allptrs─────┤
                      └─addrtaken───┘
```

where available aliasing options are:

[NO]TYPeptr	If **notypeptr** is specified, pointers to different types are never aliased. In other words, in the compilation unit, no two pointers of different types will point to the same storage location.
[NO]ALLPtrs	If **noallptrs** is specified, pointers are never aliased (this also implies **-qalias=typeptr**). Therefore, in the compilation unit, no two pointers will point to the same storage location.
[NO]ADDRtaken	If **noaddrtaken** is specified, 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.
[NO]ANSI	If **ansi** is specified, 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 (**ansi**) is the default for the **xlc xlC**, and **c89 compilers.** This option has no effect unless you also specify the **-O** option. If you select **noansi**, 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. This is the default for the **cc** compiler.

Notes

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**.

Example

To specify worst-case aliasing assumptions when compiling myprogram.c, enter:

```
xlc myprogram.c -O -qalias=noansi
```

Related References

"Compiler Command Line Options" on page 35
"#pragma disjoint" on page 269

align

▶ C ▶ C++

Purpose

Specifies what aggregate alignment rules the compiler uses for file compilation. Use this option to specify the maximum alignment to be used when mapping a class-type object, either for the whole source program or for specific parts.

Syntax

```
                          ┌─linuxppc───┐
▶▶── -q──align──=──┴─bit_packed─┴───────────────────────────────────────────▶◀
```

where available alignment options are:

linuxppc	The compiler uses default GCC alignment rules to maintain compatibility with GCC objects. This is the default.
bit_packed	The compiler uses the **bit_packed** alignment rules. This suboption is similar to the GCC **-fpack-struct** option.

See also "#pragma align" on page 261 and "#pragma options" on page 299.

Notes

If you use the **-qalign** option more than once on the command line, the last alignment rule specified applies to the file.

You can control the alignment of a subset of your code by using **#pragma align=***alignment_rule* to override the setting of the **-qalign** compiler option. Use **#pragma align=reset** to revert to a previous alignment rule. The compiler stacks alignment directives, so you can go back to using the previous alignment directive, without knowing what it is, by specifying the **#pragma align=reset** directive. For example, you can use this option if you have a class declaration within an include file and you do not want the alignment rule specified for the class to apply to the file in which the class is included.

Examples

Example 1 - Affecting Only Aggregate Definition

Using the compiler invocation:

```
xlc++ file2.C /* <-- default alignment rule for file is                */
            /*     linuxppc because no alignment rule specified */
```

Where file2.C has:

```
extern struct A A1;
typedef struct A A2;

#pragma options align=bit_packed /* <-- use bit_packed alignment rules*/
struct A {
  int a;
  char c;
};
#pragma options align=reset /* <-- Go back to default alignment rules */

struct A A1;  /* <-- aligned using bit_packed alignment rules since   */
A2 A3;        /*     this rule applied when struct A was defined      */
```

Example 2 - Imbedded #pragmas

Using the compiler invocation:

```
xlc -qalign=linuxppc file.c  /* <-- default alignment rule for file */
                             /*    is linuxppc                      */
```

Where file.c has:

```
struct A {
  int a;
  struct B {
    char c;
    double d;
#pragma options align=bit_packed /* <-- B will be unaffected by this  */
                              /*     #pragma, unlike previous behavior; */
                              /*     linuxppc alignment rules still    */
                              /*     in effect                          */
  } BB;
#pragma options align=reset /* <-- A is unaffected by this #pragma;   */
} AA;                         /*     linuxppc alignment rules still    */
                              /*     in effect                          */
```

Using the __align specifier

You can use the **__align** specifier to explicitly specify data alignment when declaring or defining a data item.

__align Specifier:

Purpose: Use the **__align** specifier to explicitly specify alignment and padding when declaring or defining data items.

Syntax:

```
declarator __align (int_const) identifier;

__align (int_const) struct_or_union_specifier [identifier] {struct_decln_list}
```

where:

int_const	Specifies a byte-alignment boundary. *int_const* must be an integer greater than 0 and equal to a power of 2.

Notes: The **__align** specifier can only be used with declarations of first-level variables and aggregate definitions. It ignores parameters and automatics.

The **__align** specifier can be used on an aggregate definition nested within another aggregate definition.

The **__align** specifier cannot be used in the following situations:

- Individual elements within an aggregate definition.
- Variables declared with incomplete type.
- Aggregates declared without definition.
- Individual elements of an array.
- Other types of declarations or definitions, such as **typedef**, **function**, and **enum**.
- Where the size of variable alignment is smaller than the size of type alignment.

Not all alignments may be representable in an object file.

Examples: Applying **__align** to first-level variables:

```
int __align(1024) varA;        /* varA is aligned on a 1024-byte boundary
                                   and padded with 1020 bytes              */
static int __align(512) varB;  /* varB is aligned on a 512-byte boundary
                                   and padded with 508 bytes               */
int __align(128) functionB( ); /* An error                                 */
typedef int __align(128) T;    /* An error                                 */
__align enum C {a, b, c};      /* An error                                 */
```

Applying **__align** to align and pad aggregate tags without affecting aggregate members:

```
__align(1024) struct structA {int i; int j;}; /* struct structA is aligned
                                                 on a 1024-byte boundary
                                                 with size including padding
                                                 of 1024 bytes            */
__align(1024) union unionA {int i; int j;}; /* union unionA is aligned
                                                 on a 1024-byte boundary
                                                 with size including padding
                                                 of 1024 bytes           */
```

Applying **__align** to a structure or union, where the size and alignment of the aggregate using the structure or union is affected:

```
__align(128) struct S {int i;};     /* sizeof(struct S) == 128            */

struct S sarray[10];                /* sarray is aligned on 128-byte boundary
                                       with sizeof(sarray) == 1280        */

struct S __align(64) svar;          /* error - alignment of variable is
                                         smaller than alignment of type   */

struct S2 {struct S s1; int a;} s2; /* s2 is aligned on 128-byte boundary
                                         with sizeof(s2) == 256           */
```

Applying **__align** to an array:

```
AnyType __align(64) arrayA[10]; /* Only arrayA is aligned on a 64-byte
                                  boundary, and elements within that array
                                  are aligned according to the alignment
                                  of AnyType.  Padding is applied after the
                                  back of the array and does not affect
                                  the size of the array member itself.   */
```

Applying **__align** where size of variable alignment differs from size of type alignment:

```
__align(64) struct S {int i;};

struct S __align(32) s1;          /* error, alignment of variable is smaller
                                     than alignment of type                */

struct S __align(128) s2;         /* s2 is aligned on 128-byte boundary     */

struct S __align(16) s3[10];      /* error                                  */
```

```
int __align(1) s4;              /* error                                  */
__align(1) struct S {int i;};   /* error                                  */
```

Related References

"Compiler Command Line Options" on page 35
"#pragma align" on page 261
"#pragma pack" on page 306

See also *Aligning data in aggregates* in the *XL C/C++ Programming Guide*.

alloca

C

Purpose

Substitutes inline code for calls to function alloca, as if **#pragma alloca** directives were in the source code.

Syntax

```
►►── -q──alloca──────────────────────────────────────────────────►◄
```

Notes

C If **#pragma alloca** is unspecified, and if you do not use **-ma**, **alloca** is treated as a user-defined identifier rather than as a built-in function.

C++ In C++ programs, you should use the **__alloca** built-in function. If your source code already references **alloca** as a function name, use the following option on the command line when invoking the compiler:

```
-Dalloca=__alloca
```

You may want to consider using a C99 variable length array in place of **alloca**.

Example

To compile myprogram.c so that calls to the function **alloca** are treated as inline, enter:

```
xlc myprogram.c -qalloca
```

Related References

"Compiler Command Line Options" on page 35
"D" on page 80
"ma" on page 170
"#pragma alloca" on page 262

altivec

C C++

Purpose

Enables compiler support for AltiVec data types.

Syntax

```
              ┌─noaltivec─┐
▶▶── -q───────┴─altivec───┴──────────────────────────────────────────▶◀
```

Notes

The AltiVec Programming Interface specification describes a special set of vector data types and operators. This option instructs the compiler to support AltiVec data types and operators.

Specifying **-qaltivec** has effect only when **-qarch** is set or implied to be **ppc970**. The compiler will ignore **-qaltivec** and issue a warning if other settings of **-qarch** are set or implied.

Also, **-qaltivec** cannot be used if **-qnoenablevmx** is in effect.

When **-qaltivec** is in effect, the following macros are defined:

- __ALTIVEC__ is defined to 1.
- __VEC__ is defined to 10205.

Example

To enable compiler support for the AltiVec Pogramming Interface specification, enter:

```
xlc myprogram.c -qaltivec
```

Related References

"Compiler Command Line Options" on page 35
"arch" on page 56
"tune" on page 242
"#pragma altivec_vrsave" on page 263

arch

C C++

Purpose

Specifies the general processor architecture for which the code (instructions) should be generated.

Syntax

```
                  ┌─ppc970─┐
►►── -q──arch──=──┴─auto───┴──────────────────────────────────────────►◄
```

where available options specify broad families of processor architectures or subgroups of those architecture families, described below.

auto
- This is implied if **-O4** or **-O5** is set or implied.
- Produces object code containing instructions that will run on the hardware platform on which it is compiled.

ppc970
- Generates instructions specific to the PowerPC 970 architecture.
- Defines the _ARCH_PPC, _ARCH_PPCGR, _ARCH_PPC970 macros.

Notes

If you want maximum performance on a specific architecture and will not be using the program on other architectures, use the appropriate architecture option.

You can use **-qarch=***suboption* with **-qtune=***suboption*. **-qarch=***suboption* specifies the architecture for which the instructions are to be generated, and **-qtune=***suboption* specifies the target platform for which the code is optimized. If **-qarch** is specified without **-qtune**, the compiler uses the default tuning option for the specified architecture, and the listing shows: `TUNE=DEFAULT`.

Example

To specify that the executable program testing compiled from myprogram.c is to run on a computer with a 32-bit PowerPCPPC970 architecture, enter:

```
xlc -o testing myprogram.c -q32 -qarch=ppc970
```

Related references

- “tune” on page 242
- “Specify Compiler Options for Architecture-Specific, 32- or 64-bit Compilation” on page 25
- “Acceptable compiler mode and processor architecture combinations” on page 343

asm

C C++

Purpose

Controls the interpretation of and subsequent generation of code for an **asm** statement.

Syntax

C The default is **-qasm=gcc**, independent of the language level. Specifying **-qasm** without a suboption is equivalent to specifying the default.

```
         ┌─asm=gcc─┐
►►── -q──┼─asm─────┼──┬────────┬──────────────────────────────────────────►◄
         └─noasm───┘  └─=gcc───┘
```

C++ The default is also**-qasm=gcc**, independent of the language level.

```
         ┌─asm=gcc─┐
►►── -q──┼─asm─────┼──┬───────────────┬───────────────────────────────────►◄
         └─noasm───┘  └─=──┬─gcc────┬─┘
                           └─stdcpp─┘
```

Notes

The **-qasm** option and its negative form control whether or not code is emitted for an **asm** statement. The positive form of the option directs the compiler to generate code for **asm** statements in the source code. The suboptions specify the syntax used to interpret the content of the **asm** statement. For example, specifying **-qasm=gcc** instructs the compiler to recognize the extended gcc syntax and semantics for **asm** statements.

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 XL-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.

Predefined macros

Whenever **asm** is treated as a keyword, the compiler predefines one of the following mutually exclusive macros, depending on the assembly language syntax specified. If assembler code is generated, the macro has the value 1; otherwise, 0.

__IBM_GCC_ASM
__IBM_STDCPP_ASM (C++ only)

Informational messages

When the option **-qinfo=eff** is also in effect, the compiler emits an informational message if no code is generated for an **asm** statement.

Whenever an **asm** statement is recognized as a valid language feature, the option **-qinfo=por** instructs the compiler to report it in an informational message. At C language levels stdc89 or stdc99, the option **-qkeyword=asm** must also be in effect.

Example

The following code snippet shows a simple use of the **-qasm** compiler option:

```
int a, b, c;
int main() {
    asm("add %0, %1, %2" : "=r"(a) : "r"(b), "r"(c) );
}
```

Related references

- "langlvl" on page 143
- "info" on page 119
- "keyword" on page 140

attr

▶ C ▶ C++

Purpose

Produces a compiler listing that includes an attribute listing for all identifiers.

Syntax

```
         ┌─noattr───────────┐
►►── -q──┴─attr──┬─────────┬┴─────────────────────────────────────────►◄
                 └─=──full─┘
```

where:

-qnoattr	Does not produce an attribute listing for identifiers in the program.
-qattr=full	Reports all identifiers in the program.
-qattr	Reports only those identifiers that are used.

See also "#pragma options" on page 299.

Notes

This option does not produce a cross-reference listing unless you also specify **-qxref**.

The **-qnoprint** option overrides this option.

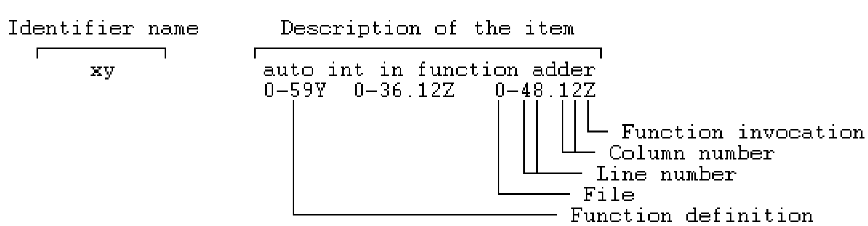
If **-qattr** is specified after **-qattr=full**, it has no effect. The full listing is produced.

Example

To compile the program myprogram.C and produce a compiler listing of all identifiers, enter:

```
xlc++ myprogram.C -qxref -qattr=full
```

A typical cross-reference listing has the form:



Related References

"Compiler Command Line Options" on page 35
"print" on page 194
"xref" on page 257
"#pragma options" on page 299

B

▶ C ▶ C++

Purpose

Determines substitute path names for programs such as the compiler, assembler, linkage editor, and preprocessor.

Syntax

```
▶▶── -B─┬────────┬──┬──────────────┬──────────────────────────────────▶◀
        └─prefix─┘  └─ -t──program─┘
```

where *program* can be any program name recognized by the **-t** compiler option. See the documentation for **t** for more information about specifying programs.

Notes

The optional *prefix* defines part of a path name to the new programs. The compiler does not add a / between the prefix and the program name.

To form the complete path name for each program, IBM XL C/C++ adds prefix to the standard program names for the compiler, assembler, linkage editor and preprocessor.

Use this option if you want to keep multiple levels of some or all of IBM XL C/C++ executables and have the option of specifying which one you want to use.

If **-B***prefix* is not specified, the default path is used.

-B -t*programs* specifies the programs to which the **-B** prefix name is to be appended.

The **-B***prefix* **-t***programs* options override the **-F***config_file* option.

Example

To compile myprogram.C using a substitute **xlc++** compiler in **/lib/tmp/mine/** enter:

```
xlc++ myprogram.C -B/lib/tmp/mine/ -tc
```

To compile myprogram.C using a substitute linkage editor in **/lib/tmp/mine/**, enter:

```
xlc++ myprogram.C -B/lib/tmp/mine/ -tl
```

Related References

"Compiler Command Line Options" on page 35
"path" on page 186
"t" on page 230

bigdata

C C++

Purpose

In 32-bit mode, allows initialized data to be larger than 16 MB in size.

Syntax

```
          ┌─nobigdata─┐
►►── -q───┴─bigdata───┴──────────────────────────────────────────────►◄
```

Notes

In 32-bit mode, the gcc size limit for initialized data is 16 MB. Use this option when creating 32-bit applications in which initialized data and call routines in shared libraries (like open, close, printf and so on) exceed 16 MB.

Related References

"Compiler Command Line Options" on page 35

bitfields

▶ C ▶ C++

Purpose

Specifies if bit fields are signed. By default, bit fields are signed.

Syntax

```
                      ┌─signed───┐
►►── -q──bitfields──=─┴─unsigned─┴───────────────────────────────►◄
```

where options are:

signed	Bit fields are signed.
unsigned	Bit fields are unsigned.

Related References

"Compiler Command Line Options" on page 35

C

C C++

Purpose

Preserves comments in preprocessed output.

Syntax

```
►►── -C──────────────────────────────────────────────────────────►◄
```

Notes

The **-C** option has no effect without either the **-E** or the **-P** option. With the **-E** option, comments are written to standard output. With the **-P** option, comments are written to an output file.

Example

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 References

"Compiler Command Line Options" on page 35
"E" on page 88
"P" on page 184

c

C C++

Purpose

Instructs the compiler to pass source files to the compiler only.

Syntax

```
▶▶── -c ──────────────────────────────────────────────────────▶◀
```

Notes

The compiled source files are not sent to the linkage editor. 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**, etc.

The **-c** option is overridden if either the **-E**, **-P**, or **-qsyntaxonly** options are specified.

The **-c** option can be used in combination with the **-o** option to provide an explicit name of the object file that is created by the compiler.

Example

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 References

"Compiler Command Line Options" on page 35
"E" on page 88
"o" on page 183
"P" on page 184
"syntaxonly" on page 229

c_stdinc

C

Purpose

Changes the standard search location for the C headers.

Syntax

```
                        ┌─:───┐
                        ▼     │
►►── -q──c_stdinc──=────path──┴─────────────────────────────────────────►◄
```

Notes

The standard search path for C headers is determined by combining the search paths specified by both this (**-qc_stdinc**) and the **-qgcc_c_stdinc** compiler option, in that order. You can find the default search path for this option in the compiler default configuration file.

If one of these compiler options is not specified or specifies an empty string, the standard search location will be the path specified by the other option. If a search path is not specified by either of the **-qc_stdinc** or **-qgcc_c_stdinc** compiler options, the default header file search path is used.

If this option is specified more than once, only the last instance of the option is used by the compiler. To specify multiple directories for a search path, specify this option once, using a **:** (colon) to separate multiple search directories.

This option is ignored if the **-qnostdinc** option is in effect.

Example

To specify **mypath/headers1** and **mypath/headers2** as being part of the standard search path, enter:

```
xlc myprogram.c -qc_stdinc=mypath/headers1:mypath/headers2
```

Related Tasks

"Directory Search Sequence for Include Files Using Relative Path Names" on page 29
"Specify Compiler Options in a Configuration File" on page 24

Related References

"Compiler Command Line Options" on page 35
"cpp_stdinc" on page 78
"gcc_c_stdinc" on page 108
"stdinc" on page 224

cache

▶ C ▶ C++

Purpose

The **-qcache** option 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──=──┬─0─────┬─┬──┴──────────────────────────────────▶◀
                         │           ├─1─────┤ │
                         │           └─n>1───┘ │
                         ├─auto────────────────┤
                         ├─cost──=──cycles─────┤
                         ├─level──=──┬─1─┬─────┤
                         │           ├─2─┤     │
                         │           └─3─┘     │
                         ├─line──=──bytes──────┤
                         ├─size──=──Kbytes─────┤
                         └─type──=──┬─C─┬──────┘
                                    ├─c─┤
                                    ├─D─┤
                                    ├─d─┤
                                    ├─I─┤
                                    └─i─┘
```

where available cache optons are:

assoc=*number*	Specifies the set associativity of the cache, where *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=*cycles*	Specifies the performance penalty resulting from a cache miss.
level=*level*	Specifies the level of cache affected, where *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 If a machine has more than one level of cache, use a separate -qcache option.
line=*bytes*	Specifies the line size of the cache.
size=*Kbytes*	Specifies the total size of the cache.

type=*cache_type* The settings apply to the specified type of cache, where *cache_type* is one of:

C or c Combined data and instruction cache

D or d Data cache

I or i Instruction cache

Notes

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.

Example

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 References

"Compiler Command Line Options" on page 35
"ipa" on page 129
"O, optimize" on page 179

chars

▶ C ▶ C++

Purpose

Instructs the compiler to treat all variables of type **char** as either **signed** or **unsigned**.

Syntax

```
                    ┌unsigned┐
▶▶── -q──chars──=───┴signed──┴──────────────────────────────────────────▶◀
```

See also "#pragma chars" on page 265 and "#pragma options" on page 299.

Notes

You can also specify sign type in your source program using either of the following preprocessor directives:

```
#pragma options chars=sign_type

#pragma chars (sign_type)
```

where *sign_type* is either **signed** or **unsigned**.

Regardless of the setting of this option, 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.

Example

To treat all **char** types as **signed** when compiling myprogram.c, enter:

```
xlc myprogram.c -qchars=signed
```

Related References

"Compiler Command Line Options" on page 35
"#pragma chars" on page 265
"#pragma options" on page 299

check

▶ C ▶ C++

Purpose

Generates code that performs certain types of runtime checking. If a violation is encountered, a runtime exception is raised by sending a **SIGTRAP** signal to the process.

Syntax

```
        ┌─nocheck─────────────────────────────┐
▶▶── -q─┴─check──┬────────────────────────────┬┴──────────────────────────▶◀
                 │       ┌─:───────────────┐  │
                 │       ▼                 │  │
                 └─=─────┬─all───────────┬─┴──┘
                         ├─nullptr───────┤
                         ├─nonullptr─────┤
                         ├─bounds────────┤
                         ├─nobounds──────┤
                         ├─divzero───────┤
                         └─nodivzero─────┘
```

where:

all	Switches on all the following suboptions. You can use the **all** option 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 options, **all** should be the first suboption.
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.
bounds | nobounds	Performs runtime checking of addresses when 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.

See also "#pragma options" on page 299.

Notes

The **-qcheck** option has several suboptions, as described above. If you use more than one *suboption*, separate each one with a colon (:).

Specifying the **-qcheck** option without any suboptions, and without any other variations of **-qcheck** on the command line, turns all of the suboptions on.

Using the **-qcheck** option with suboptions turns the specified suboptions on if they do not have the no prefix, and off if they have the no prefix.

You can specify the **-qcheck** option more than once. The suboption settings are accumulated, but the later suboptions override the earlier ones.

The **-qcheck** option affects the runtime performance of the application. When checking is enabled, runtime checks are inserted into the application, which may result in slower execution.

Examples

1. For **-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 */
}
```

2. For **-qcheck=divzero**:

```
void func3(int a, int b) {
  a / b;            /* Traps if b=0  */
}
```

Related References

"Compiler Command Line Options" on page 35
"#pragma options" on page 299

cinc

C++

Purpose

Instructs the compiler to place an **extern ″C″ { }** wrapper around the contents of an include file.

Syntax

```
          ┌─nocinc─────────────────────┐
►►── -q───┴─cinc──=──directory_prefix──┴──────────────────────────►◄
```

where:

directory_prefix	Specifies the directory where files affected by this option are found.

Notes

Include files from specified directories have the tokens **extern ″C″ {** inserted before the first statement in the include file, and **}** appended after the last statement in the include file.

Example

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();
}
```

Related References

"Compiler Command Line Options" on page 35

compact

C C++

Purpose

When used with optimization, reduces code size where possible, at the expense of execution speed.

Syntax

```
            ┌─nocompact─┐
►►── -q─────┴─compact───┴──────────────────────────────────────────►◄
```

See also “#pragma options” on page 299.

Notes

Code size is reduced by inhibiting optimizations that replicate or expand code inline, such as inlining or loop unrolling. Execution time may increase.

Example

To compile myprogram.C to reduce code size, enter:

```
xlc++ myprogram.C -O -qcompact
```

Related References

“Compiler Command Line Options” on page 35
“#pragma options” on page 299

complexgccincl

C C++

Purpose

The **-qcomplexgccincl** compiler option instructs the compiler to internally wrap **#pragma complexgcc(on)** and **#pragma complexgcc(pop)** directives around include files found in specified directories.

Syntax

```
                ┌─complexgccincl───┐   ┌─=──/usr/include─┐
►►── -q─────────┴─nocomplexgccincl─┴───┼─────────────────┼────────────────────►◄
                                       └─=──pathname─────┘
```

where:

pathname Specifies a search path for include files. If *pathname* is not specified, the compiler assumes a *pathname* of **/usr/include**.

Notes

Include files found in directories specified by the **-qcomplexgccincl** compiler option are internally wrapped by the **#pragma complexgcc(on)** and **#pragma complexgcc(pop)** directives.

Include files found in directories specified by **-qnocomplexgccincl** are not wrapped by these directives.

The default setting is **-qcomplexgccincl=/usr/include**.

Related References

"Compiler Command Line Options" on page 35
"float" on page 99
"#pragma complexgcc" on page 267

cpluscmt

▶ C

Purpose

Use this option if you want C++ comments to be recognized in C source files.

Syntax

```
►►— -q─┬─nocpluscmt─┬──────────────────────────────────────────────►◄
       └─cpluscmt───┘
```

Default

The default setting varies:

- **-qcpluscmt** is implicitly selected when you invoke the compiler with **xlc**, **xlc_r**, **cc**, or **cc_r**.
- **-qcpluscmt** is also implicitly selected when **-qlanglvl** is set to **stdc99** or **extc99**. You can override these implicit selections by specifying **-qnocpluscmt** after the **-qlanglvl** option on the command line; for example: **-qlanglvl=stdc99 -qnocpluscmt** or **-qlanglvl=extc99 -qnocpluscmt**.
- Otherwise, the default setting is **-qnocpluscmt**.

Notes

The __C99_CPLUSCMT compiler macro is defined when **cpluscmt** is selected.

The character sequence // begins a C++ comment, except within a header name, a character constant, a string literal, or a comment. Comments do not nest, and macro replacement is not performed within comments. The following character sequences are ignored within a C++ comment:

- //
- /*
- */

C++ comments have the form **//text**. The two slashes (**//**) in the character sequence must be adjacent with nothing between them. Everything to the right of them until the end of the logical source line, as indicated by a new-line character, is treated as a comment. The **//** delimiter can be located at any position within a line.

// comments are *not* part of C89. The result of the following valid C89 program will be incorrect if **-qcpluscmt** is specified:

```
main() {
  int i = 2;
  printf("%i\n", i //* 2 */
              + 1);
}
```

The correct answer is 2 (2 divided by 1). When **-qcpluscmt** is specified, the result is 3 (2 plus 1).

The preprocessor handles all comments in the following ways:

- If the **-C** option is *not* specified, all comments are removed and replaced by a single blank.
- If the **-C** option *is* specified, comments are output unless they appear on a preprocessor directive or in a macro argument.

- If **-E** is specified, continuation sequences are recognized in all comments and are output
- If **-P** is specified, comments are recognized and stripped from the output, forming concatenated output lines.

A comment can span multiple physical source lines if they are joined into one logical source line through use of the backslash (\) character. You can represent the backslash character by a trigraph (??/).

Examples

1. **Example of C++ Comments**

   The following examples show the use of C++ comments:

   ```
   // A comment that spans two \
      physical source lines

   // A comment that spans two ??/
      physical source lines
   ```

2. **Preprocessor Output Example 1**

   For the following source code fragment:

   ```
   int a;
   int b;  // A comment that spans two \
              physical source lines
   int c;
           // This is a C++ comment
   int d;
   ```

   The output for the **-P** option is:

   ```
   int a;
   int b;
   int c;

   int d;
   ```

   The C89 mode output for the **-P -C** options is:

   ```
   int a;
   int b;  // A comment that spans two    physical source lines
   int c;
           // This is a C++ comment
   int d;
   ```

   The output for the **-E** option is:

   ```
   int a;
   int b;

   int c;

   int d;
   ```

   The C89 mode output for the **-E -C** options is:

   ```
   #line 1 "fred.c"
   int a;
   int b;  // a comment that spans two \
              physical source lines
   int c;
           // This is a C++ comment
   int d;
   ```

   Extended mode output for the **-P -C** options or **-E -C** options is:

```
int a;
int b;  // A comment that spans two \
           physical source lines
int c;
        // This is a C++ comment
int d;
```

3. **Preprocessor Output Example 2 - Directive Line**

   For the following source code fragment:

```
int a;
#define mm 1   // This is a C++ comment on which spans two \
                  physical source lines
int b;
               // This is a C++ comment
int c;
```

   The output for the **-P** option is:

```
int a;
int b;

int c;
```

   The output for the **-P -C** options:

```
int a;
int b;
               // This is a C++ comment
int c;
```

   The output for the **-E** option is:

```
#line 1 "fred.c"
int a;
#line 4
int b;

int c;
```

   The output for the **-E -C** options:

```
#line 1 "fred.c"
int a;
#line 4
int b;
               // This is a C++ comment
int c;
```

4. **Preprocessor Output Example 3 - Macro Function Argument**

   For the following source code fragment:

```
#define mm(aa) aa
int a;
int b;  mm(// This is a C++ comment
              int blah);
int c;
        // This is a C++ comment
int d;
```

   The output for the **-P** option:

```
int a;
int b;  int blah;
int c;

int d;
```

   The output for the **-P -C** options:

```
int a;
int b;  int blah;
int c;
        // This is a C++ comment
int d;
```

The output for the **-E** option is:

```
#line 1 "fred.c"
int a;
int b;
int blah;
int c;

int d;
```

The output for the **-E -C** option is:

```
#line 1 "fred.c"
int a;
int b;
int blah;
int c;
        // This is a C++ comment
int d;
```

5. **Compile Example**

   To compile myprogram.c. so that C++ comments are recognized as comments, enter:

   ```
   xlc myprogram.c -qcpluscmt
   ```

Related References

"Compiler Command Line Options" on page 35
"C" on page 63
"E" on page 88
"langlvl" on page 143
"P" on page 184

cpp_stdinc

C++

Purpose

Changes the standard search location for the C++ headers.

Syntax

```
              ┌─:──────┐
              ▼        │
►►── -q──cpp_stdinc──=───path─┴──────────────────────────────────────►◄
```

Notes

The standard search path for C++ headers is determined by combining the search paths specified by both this (**-qcpp_stdinc**) and the **-qgcc_cpp_stdinc** compiler option, in that order. You can find the default search path for this option in the compiler default configuration file.

If one of these compiler options is not specified or specifies an empty string, the standard search location will be the path specified by the other option. If a search path is not specified by either of the **-qcpp_stdinc** or **-qgcc_cpp_stdinc** compiler options, the default header file search path is used.

If this option is specified more than once, only the last instance of the option is used by the compiler. To specify multiple directories for a search path, specify this option once, using a **:** (colon) to separate multiple search directories.

This option is ignored if the **-qnostdinc** option is in effect.

Example

To make **mypath/headers1** and **mypath/headers2** the standard search path, enter:

```
xlc++ myprogram.C -qcpp_stdinc=mypath/headers1:mypath/headers2
```

Related Tasks

"Directory Search Sequence for Include Files Using Relative Path Names" on page 29
"Specify Compiler Options in a Configuration File" on page 24

Related References

"Compiler Command Line Options" on page 35
"c_stdinc" on page 65
"gcc_cpp_stdinc" on page 109
"stdinc" on page 224

crt

C C++

Purpose

Instructs the linker to use the standard system startup files at link time.

Syntax

```
            ┌─crt───┐
►►── -q─────┴─nocrt─┴──────────────────────────────────────────────────►◄
```

Notes

If the **-qnocrt** compiler option is specified, the compiler will not use the standard system startup files at link time.

Related References

"Compiler Command Line Options" on page 35
"lib" on page 160

D

▶ C ▶ C++

Purpose

Defines the macro *name* as in a **#define** preprocessor directive. *definition* is an optional definition or value assigned to *name*.

Syntax

```
►►── -D──name──┬────────────────────┬──────────────────────────────────►◄
               └─=─┬──────────────┬─┘
                   └─definition───┘
```

Notes

You can also define a macro name in your source program using the **#define** preprocessor directive, provided that the macro name has not already been defined by the **-D** compiler option.

-D*name*= is equivalent to `#define` *name*.

-D*name* is equivalent to `#define` *name* `1`. (This is the default.)

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.

The **-U***name* option, which is used to undefine macros defined by the **-D** option, has a higher precedence than the **-D***name* option.

Example

1. To specify that all instances of the name `COUNT` be replaced by 100 in `myprogram.c`, enter:

   ```
   xlc myprogram.c -DCOUNT=100
   ```

   This is equivalent to having **#define COUNT 100** at the beginning of the source file.

Related References

"Compiler Command Line Options" on page 35
"U" on page 243
Appendix A, "Predefined Macros," on page 359

dataimported

C C++

Purpose

Marks data as imported.

Syntax

```
>>-- -q--dataimported--+-----------------+------------------------------><
                       |     .-:-----.   |
                       |     V       |   |
                       '-=------names-+--'
```

Notes

This option applies only to 64-bit compilations.

When this option is in effect, imported variables are dynamically bound with a shared portion of a library.

- Specifying **-qdataimported** instructs the compiler to assume that all variables are imported.
- Specifying **-qdataimported=***names* marks the named variables as being imported, where *names* is a list of variable names separated by colons (**:**). Variables not explicitly named are not affected.

Note: C++ In C++ programs, variable *names* must be specified using their mangled names. For example, assuming the following code segment:

```
struct C{
        static int i;
}
```

you would specify the variable **C::i** as being imported by specifying the compiler option in the following manner:

```
-qdataimported=i__1C
```

You can use the operating system **dump -tv** or **nm** utilities to get the mangled names from an object file. To verify a mangled name, use the **c++filt** utility.

Conflicts among the **-qdataimported** and **-qdatalocal** data-marking options are resolved in the following manner:

Options that list variable names:	The last explicit specification for a particular variable name is used.
Options that change the default:	This form does not specify a name list. The last option specified is the default for variables not explicitly listed in the name-list form.

Related References

"Compiler Command Line Options" on page 35
"datalocal" on page 82

datalocal

C C++

Purpose

Marks data as local.

Syntax

```
                         ┌─:─────┐
                         ▼       │
►►── -q──datalocal──=────names──┴──────────────────────────────────►◄
```

Notes

This option applies only to 64-bit compilation.

When this option is in effect, local variables are statically bound with the functions that use them.

- You must specify which variables are local. If no names are specified, the linker will fail to link at link-time.
- Specifying **-qdatalocal=***names* marks the named variables as local, where *names* is a list of identifiers separated by colons (**:**). Variables not explicitly named are not affected.

Note: C++ In C++ programs, variable *names* must be specified using their mangled names. For example, assuming the following code segment:

```
struct C{
        static int i;
}
```

you would specify the variable **C::i** as being local data by specifying the compiler option in the following manner:

```
-qdatalocal=i__1C
```

You can use the operating system **dump -tv** or **nm** utilities to get the mangled names from an object file. To verify a mangled name, use the **c++filt** utility.

Performance may decrease if an imported variable is assumed to be local.

Conflicts among the **-qdataimported** and **-qdatalocal** data-marking options are resolved in the following manner:

Options that list variable names:	The last explicit specification for a particular variable name is used.
Options that change the default:	This form does not specify a name list. The last option specified is the default for variables not explicitly listed in the name-list form.

Related References

"Compiler Command Line Options" on page 35
"dataimported" on page 81

dbxextra

C

Purpose

Specifies that all **typedef** declarations, **struct**, **union**, and **enum** type definitions are included for debugging.

Syntax

```
           ┌─nodbxextra─┐
►►── -q────┴─dbxextra───┴──────────────────────────────────────────────►◄
```

See also "#pragma options" on page 299.

Notes

Use this option with the **-g** option to produce additional debugging information for use with a debugger.

When you specify the **-g** option, debugging information is included in the object file. To minimize the size of object and executable files, the compiler only includes information for symbols that are referenced. Debugging information is not produced for unreferenced arrays, pointers, or file-scope variables unless **-qdbxextra** is specified.

Using **-qdbxextra** may make your object and executable files larger.

Example

To include all symbols in myprogram.c for debugging, enter:

```
xlc myprogram.c -g -qdbxextra
```

Related References

"Compiler Command Line Options" on page 35
"g" on page 107
"#pragma options" on page 299

digraph

C C++

Purpose

Lets you use digraph key combinations or keywords to represent characters not found on some keyboards.

Syntax

```
►►── -q──┬─nodigraph─┬──────────────────────────────────────────────►◄
         └─digraph───┘
```

See also "#pragma options" on page 299.

Defaults

- C **-qnodigraph** when **-qlanglvl** is set to other than **extc99** or **stdc99**.
- C **-qdigraph** when **-qlanglvl** is set to **extc99** or **stdc99**.
- C++ **-qdigraph**

Notes

A digraph is a keyword or combination of keys that lets you produce a character that is not available on all keyboards.

The digraph key combinations are:

Key Combination	Character Produced
<%	{
%>	}
<:	[
:>	]
%%	#

Additional keywords, valid in C++ programs only, are:

<table>
<tr><th>Keyword</th><th>Character Produced</th></tr>
<tr><td>bitand</td><td>&</td></tr>
<tr><td>and</td><td>&&</td></tr>
<tr><td>bitor</td><td>|</td></tr>
<tr><td>or</td><td>||</td></tr>
<tr><td>xor</td><td>^</td></tr>
<tr><td>compl</td><td>~</td></tr>
<tr><td>and_eq</td><td>&=</td></tr>
<tr><td>or_eq</td><td>|=</td></tr>
<tr><td>xor_eq</td><td>^=</td></tr>
<tr><td>not</td><td>!</td></tr>
<tr><td>not_eq</td><td>!=</td></tr>
</table>

Example

To disable digraph character sequences when compiling your program, enter:

```
xlc++ myprogram.C -qnodigraph
```

Related References

"Compiler Command Line Options" on page 35
"langlvl" on page 143
"trigraph" on page 241
"#pragma options" on page 299

directstorage

C C++

Purpose

Informs the compiler that write-through enabled or cache-inhibited storage may be referenced.

Syntax

```
         ┌nodirectstorage┐
▸▸─ -q──┴directstorage──┴──────────────────────────────────────────▸◂
```

Notes

The **-qdirectstorage** compiler option informs the compiler that write-through enabled or cache-inhibited storage may be referenced, and that appropriate compiler output should be generated.

The PowerPC architecture allows many different implementations of cache organization. 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.

Depending on the storage control attributes specified by the program and the function being performed, your program may use cache instructions to guarantee that the function is performed correctly.

For example, the **dcbz** instruction allocates a block of data in the cache and then initializes it to a series of zeroes. Though it can be used to boost performance when zeroing a large block of data, the **dcbz** instruction should be used with caution because it will cause an alignment error to occur under any of the following conditions:

- The cache block specified by the instruction is in a memory region marked cache-inhibited.
- The cache is in write-though mode.
- The L1 Dcache or L2 cache is disabled.

Specifying **-qdirectstorage** will suppress generation of the **dcbz** instruction, and avoid the alignment errors mentioned above.

Related References

"Compiler Command Line Options" on page 35

dollar

▶ C ▶ C++

Purpose

Allows the **$** symbol to be used in the names of identifiers.

Syntax

```
        ┌─nodollar─┐
▶▶── -q─┴─dollar───┴──────────────────────────────────────────────▶◀
```

When **-qdollar** is in effect, the dollar symbol `$` in an identifier is treated as a base character. If the options **-qnodollar** and **-qlanglvl=ucs** are both in effect, the dollar symbol is treated as an extended character and translated into `\u0024`.

Example

To compile myprogram.c so that **$** is allowed in identifiers in the program, enter:

```
xlc myprogram.c -qdollar
```

Related references

- "#pragma options" on page 299

E

C C++

Purpose

Instructs the compiler to preprocess the source files.

Syntax

```
▶▶── -E ──────────────────────────────────────────────────────────▶◀
```

Notes

The **-E** and **-P** options have different results. When the **-E** option is specified, the compiler assumes that the input is a C or C++ file and that the output will be recompiled or reprocessed in some way. These assumptions are:

- Original source coordinates are preserved. This is why **#line** directives are produced.
- All tokens are output in their original spelling, which, in this case, includes continuation sequences. This means that any subsequent compilation or reprocessing with another tool will give the same coordinates (for example, the coordinates of error messages).

The **-P** option is used for general-purpose preprocessing. No assumptions are made concerning the input or the intended use of the output. This mode is intended for use with input files that are not written in C or C++. As such, all preprocessor-specific constructs are processed as described in the ANSI C standard. In this case, the continuation sequence is removed as described in the "Phases of Translation" of that standard. All non-preprocessor-specific text should be output as it appears.

Using **-E** causes **#line** directives to be generated to preserve the source coordinates of the tokens. Blank lines are stripped and replaced by compensating **#line** directives.

The line continuation sequence is removed and the source lines are concatenated with the **-P** option. With the **-E** option, the tokens are output on separate lines in order to preserve the source coordinates. The continuation sequence may be removed in this case.

The **-E** option overrides the **-P**, **-o**, and **-qsyntaxonly** options, and accepts any file name.

If used with the **-M** option, **-E** will work only for files with a **.C**, **.cpp**, **.cc** (all C++ source files), **.c** (C source files), or a **.i** (preprocessed source files) filename suffix. Source files with unrecognized filename suffixes are treated and preprocessed as C files, and no error message is generated.

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, and when **-C** is not specified. Comments within a macro function argument are deleted.

The default is to preprocess, compile, and link-edit source files to produce an executable file.

Example

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 References

"Compiler Command Line Options" on page 35
"M" on page 168
"o" on page 183
"P" on page 184
"syntaxonly" on page 229

e

▶ C ▶ C++

Purpose

This option is used only together with the **-qmkshrobj** compiler option. See the description for the **-qmkshrobj** compiler option for more information.

Syntax

```
►►— -e—name————————————————————————————————————————►◄
```

Related References

"Compiler Command Line Options" on page 35
"mkshrobj" on page 178

eh

C++

Purpose

Controls whether exception handling is enabled in the module being compiled.

Syntax

```
          ┌─eh───┐
►►── -q───┴─noeh─┴──────────────────────────────────────────────►◄
```

where:

eh
: Exception handling is enabled.

noeh
: If your program does not use C++ structured exception handling, compile with **-qnoeh** to prevent generation of code that is not needed by your application.

Notes

Specifying **-qeh** also implies **-qrtti**. If **-qeh** is specified together with **-qnortti**, RTTI information will still be generated as needed.

Related References

"Compiler Command Line Options" on page 35
"rtti" on page 207

enablevmx

▶ C ▶ C++

Purpose

Enables generation of VMX (Vector Multimedia Extension) instructions.

Syntax

```
            ┌─noenablevmx─┐
▶▶── -q─────┴─enablevmx───┴──────────────────────────────────────────▶◀
```

Defaults

On operating system versions that support VMX instructions, **-qenablevmx** is set by default. Otherwise, the default is **-qnoenablevmx**.

Notes

Some processors are able to support VMX (Vector Multimedia Extension) instructions. These instructions can offer higher performance when used with algorithmic-intensive tasks such as multimedia applications.

The **-qenablevmx** compiler option enables generation of VMX instructions, but you should not use this option unless your operating system version supports VMX instructions.

Also, if **-qnoenablevmx** is in effect, **-qaltivec** cannot be used.

Related References

"Compiler Command Line Options" on page 35
"altivec" on page 55
"arch" on page 56
"hot" on page 113

enum

C C++

Purpose

Specifies the amount of storage occupied by enumerations.

Syntax

```
                      (2)
                  .-intlong-.
                  |  (1)    |
                  +-int-----+
>>-- -q--enum--=--+-small---+----------------------------------><
                  +-1-------+
                  +-2-------+
                  +-4-------+
                  '-8-------'
```

Notes:

1 C compilation default

2 C++ compilation default

where valid **enum** settings are:

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.
4	Specifies that enumerations occupy 4 bytes of storage are of type **int** if the range of enumeration values falls within the limits of **signed int**, and **unsigned int** otherwise.
8	Specifies that enumerations occupy 8 bytes of storage. In 32-bit compilation mode, the enumeration is of type **long long** if the range of enumeration values falls within the limits of **signed long long**, and **unsigned long long** otherwise. In 64-bit compilation mode, the enumeration is of type **long** if the range of enumeration values falls within the limits of **signed long**, and **unsigned long** otherwise.
int	Specifies that enumerations occupy 4 bytes of storage and are represented by **int**. Values cannot exceed the range of **signed int** in C compilations.
intlong	Specifies that enumerations will occupy 8 bytes of storage if the range of values in the enumeration exceeds the limit for **int**. See the description for **-qenum=8**. 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. See the description for **-qenum=8**

See also "#pragma enum" on page 271 and "#pragma options" on page 299.

Notes

The **-qenum=small** option allocates to an **enum** *variable* the amount of storage that is required by the smallest predefined type that can represent that range of **enum** constants. By default, an unsigned predefined type is used. If any **enum** constant is negative, a signed predefined type is used.

The **-qenum=1|2|4|8** options allocate a specific amount of storage to an **enum** *variable*. If the specified storage size is smaller than that required by the range of **enum** variables, a Severe error message is issued and compilation stops.

The ISO C 1989 and ISO C1999 Standards require that enumeration values not exceed the range of **int**. When compiling with **-qlanglvl=stdc89** or **-qlanglvl=stdc99** in effect, the compiler will behave as follows if the value of an enumeration exceeds the range of **int**:

- If **-qenum=int** is in effect, a Severe error message is issued and compilation stops.
- For all other settings of **-qenum**, an Informational message is issued and compilation continues.

The 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**.

Related References

"Compiler Command Line Options" on page 35
"#pragma enum" on page 271
"#pragma options" on page 299

Enum sizes and types – All types are signed unless otherwise noted.

	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+1) ..2147483647	ERROR[1]	int	ERROR[1]	int	int	int	long long	long long	long	long
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[2b]	ERROR[1]	long[2b]	ERROR[1]	long[2b]	long long[2b]	long long[2b]	long[2b]	long[2b]
-2^{63}..(2^{63}-1)	ERROR[1]	long[2b]	ERROR[1]	long[2b]	ERROR[1]	long[2b]	long long[2b]	long long[2b]	long[2b]	long[2b]
0..2^{64}	ERROR[1]	unsigned long[2b]	ERROR[1]	unsigned long[2b]	ERROR[1]	unsigned long[2b]	unsigned long long[2b]	unsigned long long[2b]	unsigned long[2b]	unsigned long[2b]

Notes:

1. These enumerations are too large for the **-qenum=1|2|4** settings. A Severe error is issued and compilation stops.
   To correct this condition, you should reduce the range of the enumerations, choose a larger **enum** setting, or choose a dynamic **enum** 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 compiling with **-qlanglvl=stdc89** or **-qlanglvl=stdc99** 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.

Enum sizes and types – All types are signed unless otherwise noted.

	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[2a]	ERR[2a]	long long[2b]	long long[2b]	long[2b]	long[2b]	unsigned long long[2b]	unsigned long long[2b]	unsigned long[2b]	unsigned long[2b]
-2^{63}..(2^{63}-1)	ERR[2a]	ERR[2a]	long long[2b]	long long[2b]	long[2b]	long[2b]	long long[2b]	long long[2b]	long[2b]	long[2b]
0..2^{64}	ERR[2a]	ERR[2a]	unsigned long long[2b]	unsigned long long[2b]	unsigned long[2b]	unsigned long[2b]	unsigned long long[2b]	unsigned long long[2b]	unsigned long[2b]	unsigned long[2b]

Notes:

1. These enumerations are too large for the **-qenum=1|2|4** settings. A Severe error is issued and compilation stops. To correct this condition, you should reduce the range of the enumerations, choose a larger **enum** setting, or choose a dynamic **enum** 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 compiling with **-qlanglvl=stdc89** or **-qlanglvl=stdc99** 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.

F

C C++

Purpose

Names an alternative configuration file (.cfg) for the compiler.

Syntax

```
►►── -F──┬─config_file──┬──────────┬─┬──────────────────────────────►◄
         │              └─:──stanza─┘ │
         └─:──stanza──────────────────┘
```

where suboptions are:

config_file	Specifies the name of a compiler configuration file.
stanza	Specifies the name of the command used to invoke the compiler. This directs the compiler to use the entries under *stanza* in the *config_file* to set up the compiler environment.

Notes

The default is a configuration file configured at installation time. Any file names or stanzas that you specify on the command line or within your source file override the defaults specified in the configuration file.

For information regarding the contents of the configuration file, refer to “Specify Compiler Options in a Configuration File” on page 24.

The **-B**, **-t**, and **-W** options override the **-F** option.

Example

To compile myprogram.c using a configuration file called **/usr/tmp/myvac.cfg**, enter:

```
xlc myprogram.c -F/usr/tmp/myvac.cfg:xlc
```

Related Tasks

“Specify Compiler Options in a Configuration File” on page 24

Related References

“Compiler Command Line Options” on page 35
“B” on page 60
“t” on page 230
“W” on page 253

flag

C C++

Purpose

Specifies the minimum severity level of diagnostic messages to be reported in a listing and displayed on a terminal. The diagnostic messages display with their associated sub-messages.

Syntax

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

Notes:

1 Minimum severity level messages reported in listing

2 Minimum severity level messages reported on terminal

where message severity levels are:

severity	**Description**
i	Information
w	Warning
e	Error
s	Severe error
u	Unrecoverable error

See also "#pragma options" on page 299.

Notes

You must specify a minimum message severity level for both listing and terminal reporting.

Specifying informational message levels does not turn on the **-qinfo** option.

Example

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 References

"Compiler Command Line Options" on page 35
"info" on page 119
"w" on page 254
"#pragma options" on page 299
"Compiler Messages" on page 345

float

C C++

Purpose

Specifies various floating-point options. These options provide different strategies for speeding up or improving the accuracy of floating-point calculations.

Syntax

```
               ┌─:──────────────────┐
               │ ┌─nospnans──────┐  │
               │ ├─norsqrt───────┤  │
               │ ├─norrm─────────┤  │
               │ ├─nonans────────┤  │
               │ ├─maf───────────┤  │
               │ ├─fold──────────┤  │
               │ ├─nofltint──────┤  │
               │ ├─complexgcc────┤  │
               ▼ │               │  │
►►── -qfloat──=──┼─nocomplexgcc──┼──┴──────────────────────────────────────►◄
                 ├─fltint────────┤
                 ├─nofold────────┤
                 ├─nomaf─────────┤
                 ├─nans──────────┤
                 ├─rrm───────────┤
                 ├─rsqrt─────────┤
                 └─spnans────────┘
```

Option selections are described in the **Notes** section below. See also "#pragma options" on page 299.

Notes

Using **float** suboptions other than the default settings may produce varying results in floating point computations. Incorrect computational results may be produced if not all required conditions for a given suboption are met. For these reasons, 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.

You can specify one or more of the following **float** suboptions.

complexgcc nocomplexgcc	Enables compatibility with GCC passing and returning of **_complex**. The default is **float=complexgcc** when compiling in 32-bit mode, and **float=nocomplexgcc** when compiling in 64-bit mode.

fltint nofltint	Speeds up floating-point-to-integer conversions by using faster inline code that does not check for overflows. The default is **float=nofltint**, which checks floating-point-to-integer conversions for out-of-range values. This suboption must only be used with an optimization option. • With **-O2** in effect, **-qfloat=nofltint** is the implied setting. • With **-O3** and greater in effect, **-qfloat=fltint** is implied. To include range checking in floating-point-to-integer conversions with the **-O3** option, specify **-qfloat=nofltint**. • **-qnostrict** sets **-qfloat=fltint** Changing the optimization level will not change the setting of the **fltint** suboption if **fltint** has already been specified. If the **-qstrict | -qnostrict** and **-qfloat=** options conflict, the last setting is used.
fold nofold	Specifies that constant floating-point expressions are to be evaluated at compile time rather than at run time.
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 option may affect the precision of floating-point intermediate results.
nans nonans	Generates extra instructions to detect signalling NaN (Not-a-Number) when converting from single-precision to double-precision at run time. The option **nonans** specifies that this conversion need not be detected. **-qfloat=nans** is required for full compliance to the IEEE 754 standard. When used with the **-qflttrap** or **-qflttrap=invalid** option, the compiler detects invalid operation exceptions in comparison operations that occur when one of the operands is a signalling NaN.
rrm norrm	Prevents floating-point optimizations that are incompatible with runtime rounding to plus and minus infinity modes. Informs the compiler that the floating-point rounding mode may change at run time or that the floating-point rounding mode is not *round to nearest* at run time. **-qfloat=rrm** must be specified if the Floating Point Status and Control register is changed at run time (as well as for initializing exception trapping).

rsqrt norsqrt	Specifies whether a sequence of code that involves division by the result of a square root can be replaced by calculating the reciprocal of the square root and multiplying. Allowing this replacement produces code that runs faster. • For **-O2**, the default is **-qfloat=norsqrt**. • For **-O3**, the default is **-qfloat=rsqrt**. Use **-qfloat=norsqrt** to override this default. • **-qnostrict** sets **-qfloat=rsqrt**. (Note that **-qfloat=rsqrt** means that **errno** will *not* be set for any **sqrt** function calls.) • **-qfloat=rsqrt** has no effect unless **-qignerrno** is also specified. Changing the optimization level will not change the setting of the **rsqrt** option if **rsqrt** has already been specified. If the **-qstrict | -qnostrict** and **-qfloat=** *options* conflict, the last setting is used.
spnans nospnans	Generates extra instructions to detect signalling NaN on conversion from single-precision to double-precision. The option **nospnans** specifies that this conversion need not be detected.

Example

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 references

- “arch” on page 56
- “complexgccincl” on page 73
- “flttrap” on page 102
- “strict” on page 225
- “#pragma complexgcc” on page 267
- “#pragma options” on page 299

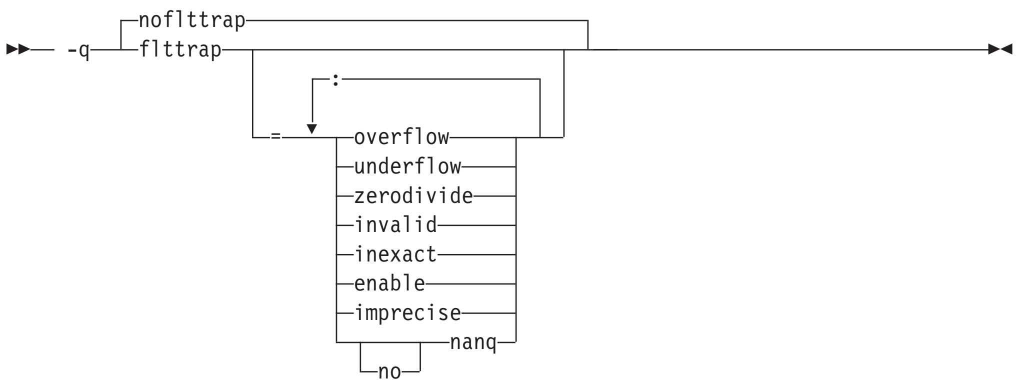
flttrap

C C++

Purpose

Generates extra instructions to detect and trap run-time floating-point exceptions.

Syntax



where suboptions do the following:

ENable	Enables the specified exceptions in the prologue of the main program. With the exception of **nanq** (described below), this suboption is required if you want to turn on exception trapping options listed below without modifying the source code.
OVerflow	Generates code to detect and trap floating-point overflow.
UNDerflow	Generates code to detect and trap floating-point underflow.
ZEROdivide	Generates code to detect and trap floating-point division by zero.
INValid	Generates code to detect and trap floating-point invalid operation exceptions.
INEXact	Generates code to detect and trap floating-point inexact exceptions.
IMPrecise	Generates code for imprecise detection of the specified exceptions. If an exception occurs, it is detected, but the exact location of the exception is not determined.
nanq	Generates code to detect and trap NaNQ (Not a Number Quiet) exceptions handled by or generated by floating point operations. The **nanq** and **nonanq** settings are not affected by **-qnoflttrap**, **-qflttrap**, or **-qflttrap=enable**.

See also "#pragma options" on page 299.

Notes

This option is recognized during linking. **-qnoflttrap** specifies that these extra instructions need not be generated.

Specifying the **-qflttrap** option with no suboptions is equivalent to setting **-qflttrap=overflow:underflow:zerodivide:invalid:inexact**. The exceptions are not automatically enabled, and all floating-point operations are checked to provide precise exception-location information.

If specified with **#pragma options**, the **-qnoflttrap** option *must* be the first option specified.

If your program contains signalling NaNs, you should use the **-qfloat=nans** 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 **-qoptimize** options:

- 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.

Example

To compile myprogram.c so that floating-point overflow and underflow and divide by zero are detected, enter:

```
xlc myprogram.c -qflttrap=overflow:underflow:zerodivide:enable
```

Related References

"Compiler Command Line Options" on page 35
"float" on page 99
"O, optimize" on page 179

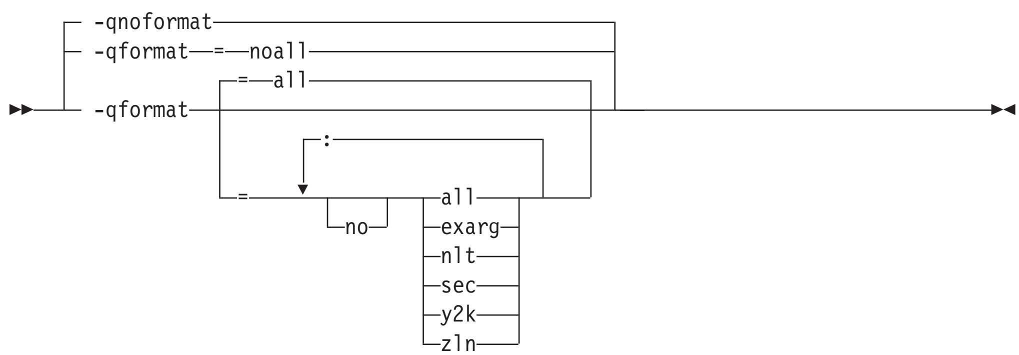
format

C C++

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



where suboptions are:

- **all** Turns on all format diagnostic messages.
- **exarg** Warns if excess arguments appear in printf and scanf style function calls.
- **nlt** Warns if a format string is not a string literal, unless the format function takes its format arguments as a va_list.
- **sec** Warns of possible security problems in use of format functions.
- **y2k** Warns of *strftime* formats that produce a 2-digit year.
- **zln** Warns of zero-length formats.

Note: Specifying **no** in front of any of the above suboptions disables that group of diagnostic messages. For example, to turn off diagnostic messages for y2k warnings, specify **-qformat=noy2k** on the command line.

Notes

If **-qformat** is *not* specified on the command line, the compiler assumes a default setting of **-qnoformat**, which is equivalent to **-qformat=noall**.

If **-qformat** is specified on the command line without any suboptions, the compiler assumes a default setting of **-qformat=all**.

Examples

1. To enable all format string diagnostics, enter either of the following:
   ```
   xlc++ myprogram.C -qformat=all
   xlc++ myprogram.C -qformat
   ```
2. To enable all format diagnostic checking except that for y2k date diagnostics, enter:
   ```
   xlc++ myprogram.C -qformat=all:noy2k
   ```

Related References

"Compiler Command Line Options" on page 35

fullpath

C C++

Purpose

Specifies what path information is stored for files when you use the **-g** compiler option.

Syntax

```
            ┌─nofullpath─┐
►►── -q─────┴─fullpath───┴─────────────────────────────────────────────►◄
```

Notes

Using **-qfullpath** causes the compiler to preserve the absolute (full) path name of source files specified with the **-g** option.

The relative path name of files is preserved when you use **-qnofullpath**.

-qfullpath is useful if the executable file was moved to another directory. If you specified **-qnofullpath**, the debugger would be unable to find the file unless you provide a search path in the debugger. Using **-qfullpath** would locate the file successfully.

Related References

"Compiler Command Line Options" on page 35
"g" on page 107

funcsect

▶ C ▶ C++

Purpose

Places instructions for each function in a separate object file control section or csect. By default, each object file will consist of a single control section combining all functions defined in the corresponding source file.

Syntax

```
           ┌─nofuncsect─┐
▶▶── -q────┴─funcsect───┴──────────────────────────────────────────▶◀
```

Notes

Using multiple csects can reduce the size of the final executable by allowing the linkage editor to remove functions that are not called or that have been inlined by the optimizer at all places they are called.

If this option is specified in **#pragma options**, the pragma directive must be specified before the first statement in the compilation unit.

Related References

"Compiler Command Line Options" on page 35
"#pragma options" on page 299

g

C C++

Purpose

Generates information used for debugging tools such as the GNU GDB Debugger.

Syntax

```
►►── -g──────────────────────────────────────────────────────────────►◄
```

Notes

Specifying **-g** will turn off all inlining unless you explicitly request it. For example:

Options	Effect on inlining
-g	No inlining.
-O	Inline declared functions.
-O -Q	Inline declared functions and auto inline others.
-g -O	Inline declared functions.
-g -O -Q	Inline declared functions and auto inline others.

The default with **-g** is not to include information about unreferenced symbols in the debugging information.

To include information about both referenced and unreferenced symbols, use the **-qdbxextra** option with **-g**.

To specify that source files used with **-g** are referred to by either their absolute or their relative path name, use **-qfullpath**.

You can also use the **-qlinedebug** option to produce abbreviated debugging information in a smaller object size.

Example

To compile myprogram.c to produce an executable program testing so you can debug it, enter:

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

To compile myprogram.c to produce an executable program named testing_all, and containing additional information about unreferenced symbols so you can debug it, enter:

```
xlc myprogram.c -o testing_all -g -qdbxextra
```

Related references

- “O, optimize” on page 179
- “Q” on page 199

gcc_c_stdinc

C

Purpose

Changes the standard search location for the gcc headers.

Syntax

```
                            ┌─:───┐
                            ▼     │
►►── -qgcc_c_stdinc──=──────path──┴──────────────────────────────────────►◄
```

Notes

The standard search path for gcc headers is determined by combining the search paths specified by both the **-qc_stdinc** and this (**-qgcc_c_stdinc**) compiler option, in that order. You can find the default search path for this option in the compiler default configuration file.

If one of these compiler options is not specified or specifies an empty string, the standard search location will be the path specified by the other option. If a search path is not specified by either of the **-qc_stdinc** or **-qgcc_c_stdinc** compiler options, the default header file search path is used.

If this option is specified more than once, only the last instance of the option is used by the compiler. To specify multiple directories for a search path, specify this option once, using a **:** (colon) to separate multiple search directories.

This option is ignored if the **-qnostdinc** option is in effect.

Example

To specify **mypath/headers1** and **mypath/headers2** as being part of the standard search path, enter:

```
xlc myprogram.c -qgcc_c_stdinc=mypath/headers1:mypath/headers2
```

Related Tasks

"Directory Search Sequence for Include Files Using Relative Path Names" on page 29
"Specify Compiler Options in a Configuration File" on page 24

Related References

"Compiler Command Line Options" on page 35
"c_stdinc" on page 65
"cpp_stdinc" on page 78
"gcc_cpp_stdinc" on page 109
"stdinc" on page 224

gcc_cpp_stdinc

C++

Purpose

Changes the standard search location for the g++ headers.

Syntax

```
                           ┌─:──┐
                           ▼    │
►►── -qgcc_cpp_stdinc──=────path─┴──────────────────────────────────►◄
```

Notes

The standard search path for g++ headers is determined by combining the search paths specified by both the **-qcpp_stdinc** and this (**-qgcc_cpp_stdinc**) compiler option, in that order. You can find the default search path for this option in the compiler default configuration file.

If one of these compiler options is not specified or specifies an empty string, the standard search location will be the path specified by the other option. If a search path is not specified by either of the **-qcpp_stdinc** or **-qgcc_cpp_stdinc** compiler options, the default header file search path is used.

If this option is specified more than once, only the last instance of the option is used by the compiler. To specify multiple directories for a search path, specify this option once, using a **:** (colon) to separate multiple search directories.

This option is ignored if the **-qnostdinc** option is in effect.

Example

To specify **mypath/headers1** and **mypath/headers2** as being part of the standard search path, enter:

```
xlc++ myprogram.C -qgcc_cpp_stdinc=mypath/headers1:mypath/headers2
```

Related Tasks

"Directory Search Sequence for Include Files Using Relative Path Names" on page 29
"Specify Compiler Options in a Configuration File" on page 24

Related References

"Compiler Command Line Options" on page 35
"c_stdinc" on page 65
"cpp_stdinc" on page 78
"gcc_c_stdinc" on page 108
"stdinc" on page 224

genproto

C

Purpose

Produces ANSI prototypes from K&R function definitions. This should help to ease the transition from K&R to ANSI.

Syntax

```
           ┌─nogenproto────────────────┐
►►── -q────┴─genproto──┬──────────────┬┴──────────────────────────────────►◄
                       └─=──parmnames─┘
```

Notes

Using **-qgenproto** without **parmnames** will cause prototypes to be generated without parameter names. Parameter names are included in the prototype when **parmnames** is specified.

Example

For the following function, foo.c:

```
foo(a,b,c)
  float a;
  int *b;
  int c;
```

specifying

```
xlc -c -qgenproto foo.c
```

produces

```
int foo(double, int*, int);
```

The parameter names are dropped. On the other hand, specifying

```
xlc -c -qgenproto=parmnames foo.c
```

produces

```
int foo(double a, int* b, int c);
```

In this case the parameter names are kept.

Note that **float a** is represented as **double** or **double a** in the prototype, since ANSI states that all narrow-type arguments (such as **char**s, **short**s, and **float**s) are widened before they are passed to K&R functions.

Related References

"Compiler Command Line Options" on page 35

halt

C C++

Purpose

Instructs the compiler to stop after the compilation phase when it encounters errors of specified *severity* or greater.

Syntax

```
                   (2)
                 ┌─────e─┐
                   (1)
                 ├─────s─┤
►►── -qhalt──=───┼─i─────┼──────────────────────────────────────────►◄
                 ├─w─────┤
                 └─u─────┘
```

Notes:

1 Default for C++ compilations.

2 Default for C compilations.

where severity levels in order of increasing severity are:

severity	**Description**
i	Information
w	Warning
e	Error (C only)
s	Severe error
u	Unrecoverable error

See also "#pragma options" on page 299.

Notes

When the compiler stops as a result of the **-qhalt** option, the compiler return code is nonzero.

When **-qhalt** is specified more than once, the lowest severity level is used.

The **-qhalt** option can be overridden by the **-qmaxerr** option.

Diagnostic messages may be controlled by the **-qflag** option.

Example

To compile myprogram.c so that compilation stops if a **warning** or higher level message occurs, enter:

```
xlc myprogram.c -qhalt=w
```

Related References

"Compiler Command Line Options" on page 35
"flag" on page 98
"maxerr" on page 173
"#pragma options" on page 299

haltonmsg

C++

Purpose

Instructs the compiler to stop after the compilation phase when it encounters the specified *msg_number*.

Syntax

```
                     ┌─,──────────┐
                     ▼            │
►►── -qhaltonmsg──=────msg_number─┴──────────────────────────────►◄
```

Notes

When the compiler stops as a result of the **-qhaltonmsg** option, the compiler return code is nonzero.

You can specify more than one message number with the **-qhaltonmsg** option. Additional message numbers must be separated by a comma.

Related References

"Compiler Command Line Options" on page 35
"Compiler Messages" on page 345

hot

C C++

Purpose

Instructs the compiler to perform **h**igh-**o**rder loop analysis and **t**ransformations during optimization.

Syntax

```
           ┌─nohot──────────────────────────────────┐
►►── -q ───┴─hot───┬────────────────────────────┬──┴──────────────────────►◄
                   │      ┌─vector──────────┐    │
                   │      ├─simd────────────┤    │
                   └─ = ──┼─nosimd──────────┼────┘
                          ├─novector────────┤
                          └─arraypad─┬──────┬┘
                                     └─=─n─┘
```

where:

arraypad
: 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.

arraypad=*n*
: The compiler will pad every array in the code. The pad amount must be a positive integer value, and each array will be padded by an integral number of elements. Because *n* is an integral value, we recommend that pad values be multiples of the largest array element size, typically 4, 8, or 16.

simd | nosimd
: The compiler converts certain operations that are performed in a loop on successive elements of an array into a call to a VMX (Vector Multimedia Extension) instruction. This call calculates several results at one time, which is faster than calculating each result sequentially.

  If you specify **-qhot=nosimd**, the compiler performs optimizations on loops and arrays, but avoids replacing certain code with calls to VMX instructions.

  This suboption has effect only when **-qarch=ppc970** is in effect, and your operating system is of a version that supports VMX instructions.

vector | novector
: The compiler converts 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 library routine. This call will calculate several results at one time, which is faster than calculating each result sequentially.

  If you specify **-qhot=novector**, the compiler performs high-order transformations on loops and arrays, but avoids optimizations where certain code is replaced by calls to vector library routines. The **-qhot=vector** option may affect the precision of your program's results so you should specify either **-qhot=novector** or **-qstrict** if the change in precision is unacceptable to you.

Default

The **-qhot=simd** and **-qhot=vector** suboptions are on by default when you specify the **-qhot**, **-O4**, or **-O5** options.

Notes

If you do not also specify optimization of at least level 2 when specifying **-qhot** on the command line, the compiler assumes **-O2**.

Because of the implementation of the cache architecture, array dimensions that are powers of two can lead to decreased cache utilization. The optional **arraypad** suboption permits the compiler to increase the dimensions of arrays where doing so might improve the efficiency of array-processing loops. If you have large arrays with some dimensions (particularly the first one) that are powers of 2, or if you find that your array-processing programs are slowed down by cache misses or page faults, consider specifying **-qhot=arraypad**.

The **simd** suboption optimizes array data to run mathematical operations in parallel where the target architecture allows such operations. Parallel operations occur in 16–byte vector registers. The compiler divides vectors that exceed the register length into 16-byte units to facilitate optimization. A 16-byte unit can contain one of the following types of data:

- 4 Integers.
- 8 2–byte units
- 16 1–byte units

Short vectorization does not support double-precision floating point mathematics, and you must specify **-qarch=ppc970**. Applying **-qhot=simd** optimization is useful for applications with significant image processing demands.

The **vector** suboption optimizes array data to run mathematical operations in parallel where applicable. The compiler uses standard registers with no vector size restrictions. The **vector** suboption supports single and double-precision floating-point mathematics, and is useful for applications with significant mathematical processing deamnds.

Both **-qhot=arraypad** and **-qhot=arraypad=*n*** are unsafe options. They do not perform any checking for reshaping or equivalences that may cause the code to break if padding takes place.

Example

The following example turns on the **-qhot=arraypad** option:

```
xlc -qhot=arraypad myprogram.c
```

Related References

"Compiler Command Line Options" on page 35
"C" on page 63
"O, optimize" on page 179

I

C C++

Purpose

Specifies an additional search path for **#include** filenames that do not specify an absolute path.

Syntax

```
►►── -I──directory──────────────────────────────────────────────►◄
```

Notes

The value for *directory* must be a valid path name (for example, **/u/golnaz**, or **/tmp**, or **./subdir**). The compiler appends a slash (/) to the directory and then concatenates it with the file name before doing the search. The path directory is the one that the compiler searches first for **#include** files whose names do not start with a slash (/). If directory is not specified, the default is to search the standard 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. See *Directory Search Sequence for Include Files Using Relative Path Names* for more information about searching directories.

If you specify a full (absolute) path name on the **#include** directive, this option has no effect.

Example

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 Tasks

"Directory Search Sequence for Include Files Using Relative Path Names" on page 29

Related References

"Compiler Command Line Options" on page 35

idirfirst

▶ C ▶ C++

Purpose

Specifies the search order for files included with the **#include "***file_name***"** directive.

Syntax

```
            ┌─noidirfirst─┐
▶▶── -q ──┴─idirfirst───┴──────────────────────────────────────────▶◀
```

See also "#pragma options" on page 299.

Notes

Use **-qidirfirst** with the **-I** option.

The normal search order (for files included with the **#include "***file_name***"** directive) *without* the **idirfirst** option is:

1. Search the directory where the current source file resides.
2. Search the directory or directories specified with the **-I** option.
3. Search the standard include directories.

With **-qidirfirst**, the directories specified with the **-I** option are searched before the directory where the current file resides.

-qidirfirst has no effect on the search order for the **#include <file_name>** directive.

-qidirfirst is independent of the **-qnostdinc** option, which changes the search order for both **#include "file_name"** and **#include <file_name>**.

The search order of files is described in *Directory Search Sequence for Include Files Using Relative Path Names*.

The last valid **#pragma options** [**no**]**idirfirst** remains in effect until replaced by a subsequent **#pragma options** [**no**]**idirfirst**.

Example

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 Tasks

"Directory Search Sequence for Include Files Using Relative Path Names" on page 29

Related References

"Compiler Command Line Options" on page 35
"I" on page 115
"stdinc" on page 224
"#pragma options" on page 299

ignerrno

C C++

Purpose

Allows the compiler to perform optimizations that assume **errno** is not modified by system calls.

Syntax

```
            ┌─noignerrno─┐
►►── -q─────┴─ignerrno───┴──────────────────────────────────────►◄
```

See also "#pragma options" on page 299.

Notes

Some system library routines set **errno** when an exception occurs. This setting and subsequent side effects of **errno** may be ignored by specifying **-qignerrno**.

Specifying a **-O3** or greater optimization option will also set **-qignerrno**. If you require both optimization and the ability to set **errno**, you should specify **-qnoignerrno** after the optimization option on the command line.

Related References

"Compiler Command Line Options" on page 35
"O, optimize" on page 179
"#pragma options" on page 299

ignprag

▶ C ▶ C++

Purpose

Instructs the compiler to ignore certain pragma statements.

Syntax

```
>>-- -qignprag--=--+-disjoint-+--------------------------------------><
                   +-isolated-+
                   +-all------+
                   '-omp------'
```

where pragma statements affected by this option are:

disjoint	Ignores all **#pragma disjoint** directives in the source file.
isolated	Ignores all **#pragma isolated_call** directives in the source file.
all	Ignores all **#pragma isolated_call** and **#pragma disjoint** directives in the source file.
omp	Ignores all OpenMP parallel processing directives in the source file, such as **#pragma omp parallel**, **#pragma omp critical**.

See also "#pragma options" on page 299.

Notes

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 **-qignprag** with the **-O** option, the information specified in the aliasing pragmas is likely incorrect.

Example

To compile myprogram.c and ignore any #pragma isolated_call directives, enter:

```
xlc myprogram.c -qignprag=isolated
```

Related References

"Compiler Command Line Options" on page 35
"#pragma disjoint" on page 269
"#pragma isolated_call" on page 287
"#pragma options" on page 299
"Pragmas to Control Parallel Processing" on page 323

info

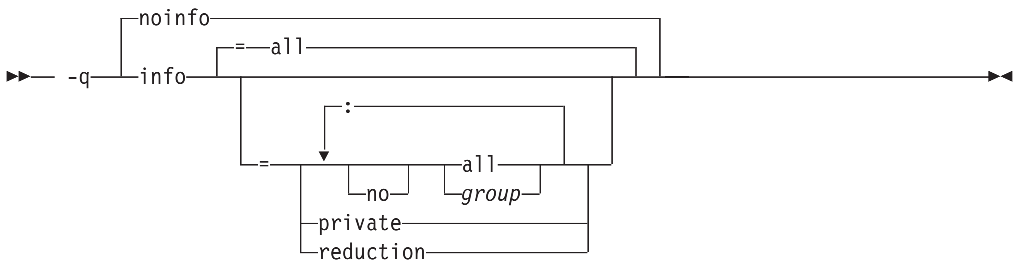
C C++

Purpose

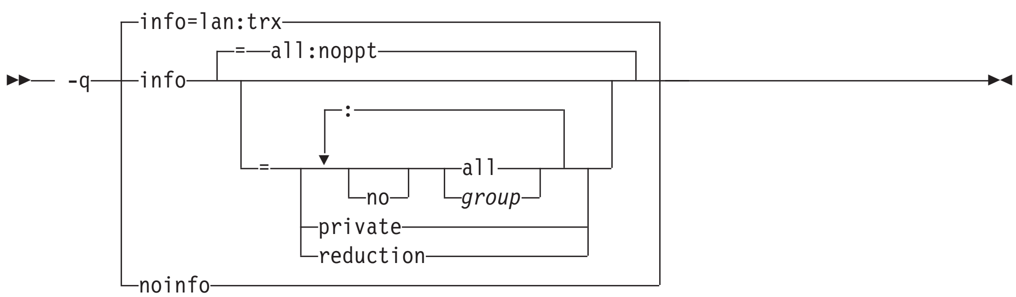
Produces informational messages.

Syntax

C



C++



where **-qinfo** options and diagnostic message groups are described in the **Notes** section below.

See also "#pragma info" on page 281 and "#pragma options" on page 299.

Defaults

If you do not specify **-qinfo** on the command line, the compiler assumes:

1. C **-qnoinfo**
2. C++ **-qinfo=lan:trx**

If you specify **-qinfo** on the command line without any suboptions, the compiler assumes:

1. C **-qinfo=all**
2. C++ **-qinfo=all:noppt**

Notes

Specifying **-qinfo=all** or **-qinfo** with no suboptions turns on all diagnostic messages for all groups except for the **ppt** (preprocessor trace) group in C++ code.

Specifying **-qnoinfo** or **-qinfo=noall** turns off all diagnostic messages for all groups.

You can use the **#pragma options info=***suboption*[**:***suboption* ...] or **#pragma options noinfo** forms of this compiler option to temporarily enable or disable messages in one or more specific sections of program code.

Available forms of the **-qinfo** option are:

all
: Turns on all diagnostic messages for all groups.

  C The **-qinfo** and **-qinfo=all** forms of the option have the same effect.

  C++ The **-qinfo** and **-qinfo=all** forms of the option both have the same effect, but do not include the **ppt** group (preprocessor trace).

lan
: Enables diagnostic messages informing of language level effects. This is the default for C++ compilations.

noall
: Turns off all diagnostic messages for specific portions of your program.

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 — Turns on or off specific groups of messages, where *group* can be one or more of:

group	Type of messages returned or suppressed
c99|noc99	C code that may behave differently between C89 and C99 language levels.
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.
inl|noinl	Functions not inlined.
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.
vft|novft	Generation of virtual function tables in C++ programs.
zea|nozea	Zero-extent arrays.

Example

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
```

Related references

- “haltonmsg” on page 112
- “suppress” on page 227

initauto

C C++

Purpose

Initializes automatic variables to the two-digit hexadecimal byte value *hex_value*.

Syntax

```
         ┌─noinitauto──────────┐
►►── -q──┴─initauto──=──hex_value─┴──────────────────────────────────►◄
```

See also "#pragma options" on page 299.

Notes

The 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.

There is no default setting for the initial value of **-qinitauto**. You must set an explicit value (for example, **-qinitauto=FA**).

Example

To compile myprogram.c so that automatic variables are initialized to hex value FF (decimal 255), enter:

```
xlc myprogram.c -qinitauto=FF
```

Related References

"Compiler Command Line Options" on page 35

inlglue

C C++

Purpose

Generates fast external linkage by inlining the pointer glue code necessary to make a call to an external function or a call through a function pointer.

Syntax

```
           ┌noinlglue┐
►►── -q────└inlglue──┘─────────────────────────────────────────────►◄
```

See also "#pragma options" on page 299.

Notes

This option applies only to 64-bit compilation.

Glue code, generated by the linker, is used for passing control between two external functions, or when you call functions through a pointer. Therefore 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.

On selected architectures, the inlining of glue code is automated through the selection of hardware performance tuning options. Specifying **-q64** and **-qtune=pwr4**,**-qtune=pwr5**, **-qtune=ppc970**, or **-qtune=auto** on a system that uses one of these architectures, automatically enables **-qinlglue**.

Inlining glue code can cause the code size to grow. The option **-qcompact** reduces code size, but it should be noted that **-qcompact** overrides **-qinlglue**, regardless of other options specified.

Related references

- "#pragma options" on page 299

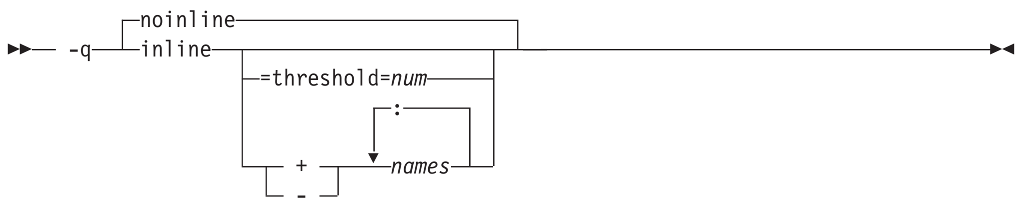
inline

C C++

Purpose

Attempts to inline functions instead of generating calls to those functions. Inlining is performed if possible but, depending on which optimizations are performed, some functions might not be inlined.

Syntax



C The following **-qinline** options apply in the C language:

-qinline	The compiler attempts to inline all appropriate functions with 20 executable source statements or fewer, subject to any other settings of the suboptions to the **-qinline** option. If **-qinline** is specified last, all functions are inlined.
-qinline=threshold=*num*	Sets a size limit on the functions to be inlined. The number of executable statements must be less than or equal to *num* for the function to be inlined. *num* must be a positive integer. The default value is 20. Specifying a threshold value of **0** causes no functions to be inlined except those functions marked with the **__inline**, **_Inline**, or **_inline** keywords.

The *num* value applies to logical C statements. 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 */
   }
}
```

-qinline-*names*

The compiler does not inline functions listed by *names*. Separate each *name* with a colon (:). All other appropriate functions are inlined. The option implies **-qinline**.

For example:

```
-qinline-salary:taxes:expenses:benefits
```

causes all functions except those named **salary**, **taxes**, **expenses**, or **benefits** to be inlined if possible.

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

-qinline+*names*	Attempts to inline the functions listed by *names* and any other appropriate functions. Each *name* must be separated by a colon (**:**). The option implies **-qinline**.
	
	For example,
	`-qinline+food:clothes:vacation`
	
	causes all functions named **food**, **clothes**, or **vacation** to be inlined if possible, along with any other functions eligible for inlining.
	
	A warning message is issued for functions that are not defined in the source file or that are defined but cannot be inlined.
	
	This suboption overrides any setting of the *threshold* value. You can use a threshold value of zero along with **-qinline+***names* to inline specific functions. For example:
	`-qinline=threshold=0`
	
	followed by:
	`-qinline+salary:taxes:benefits`
	
	causes *only* the functions named **salary**, **taxes**, or **benefits** to be inlined, if possible, and no others.

-qnoinline	Does not inline any functions. If **-qnoinline** is specified last, no functions are inlined.

C++ The following **-qinline** options apply to the C++ language:

`-qinline`	Compiler inlines all functions that it can.
`-qnoinline`	Compiler does not inline any functions.

Default

The default is to treat inline specifications as a hint to the compiler, and the result depends on other options that you select:

- If you optimize your program using one of the **-O** compiler options, the compiler attempts to inline all functions declared as inline. Otherwise, the compiler attempts to inline only some of the simpler functions declared as inline.

Notes

The **-qinline** option is functionally equivalent to the **-Q** option.

If you specify the **-g** option (to generate debug information), inlining may be affected. See the information for the **-g** compiler option.

Because inlining does not always improve run time, you should test the effects of this option on your code. Do not attempt to inline recursive or mutually recursive functions.

Normally, application performance is optimized if you request optimization (**-O** option), and compiler performance is optimized if you do not request optimization.

To maximize inlining, specify optimization (**-O**) and also specify the appropriate **-qinline** options.

The XL C/C++ (**inline**, **_inline**, **_Inline**, and **__inline**) C language keywords override all **-qinline** options except **-qnoinline**. The compiler will try to inline functions marked with these keywords regardless of other **-qinline** option settings.

The inline, _Inline, _inline, and __inline Function Specifiers: The compiler provides a set of keywords that you can use to specify functions that you want the compiler to inline. The keywords, which are functionally equivalent, are:

- inline (C99 and C++ only)
- _Inline (C only)
- _inline (C only)
- __inline (C and C++)

For example:

```
_Inline int catherine(int a);
```

suggests to the compiler that function `catherine` be inlined, meaning that code is generated for the function, rather than a function call.

Using the inline specifiers with data or to declare the main() function generates an error.

By default, function inlining is turned off, and functions qualified with inline specifiers are treated simply as extern functions. To turn on function inlining, specify either the **-qinline** or **-Q** compiler options. Inlining is also turned on if you turn optimization on with the **-O** or **-qoptimize** compiler option.

Recursive functions (functions that call themselves) are inlined for the first occurrence only. The call to the function from within itself is not inlined.

You can also use the **-qinline** or **-Q** compiler options to automatically inline all functions smaller than a specified size. For best performance, however, use the inline keywords to choose the functions you want to inline rather than using automatic inlining.

An inline function can be declared and defined simultaneously. If it is declared with one of the inline keywords, it can be defined without any inline keywords. The following code fragment shows an inline function definition. Note that the inline keywords are specified for both a function definition and function declaration.

```
_inline int add(int i, int j) { return i + j; }

inline double fahr(double t);
```

Note: The use of the inline specifier does not change the meaning of the function, but inline expansion of a function may not preserve the order of evaluation of the actual arguments.

Example

To compile myprogram.C so that no functions are inlined, enter:

```
xlc++ myprogram.C -O -qnoinline
```

To compile `myprogram.c` so that the compiler attempts to inline functions of fewer than 12 lines, enter:

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

Related references

- "O, optimize" on page 179
- "Q" on page 199
- "g" on page 107

ipa

▶ C ▶ C++

Purpose

Turns on or customizes a class of optimizations known as interprocedural analysis (IPA).

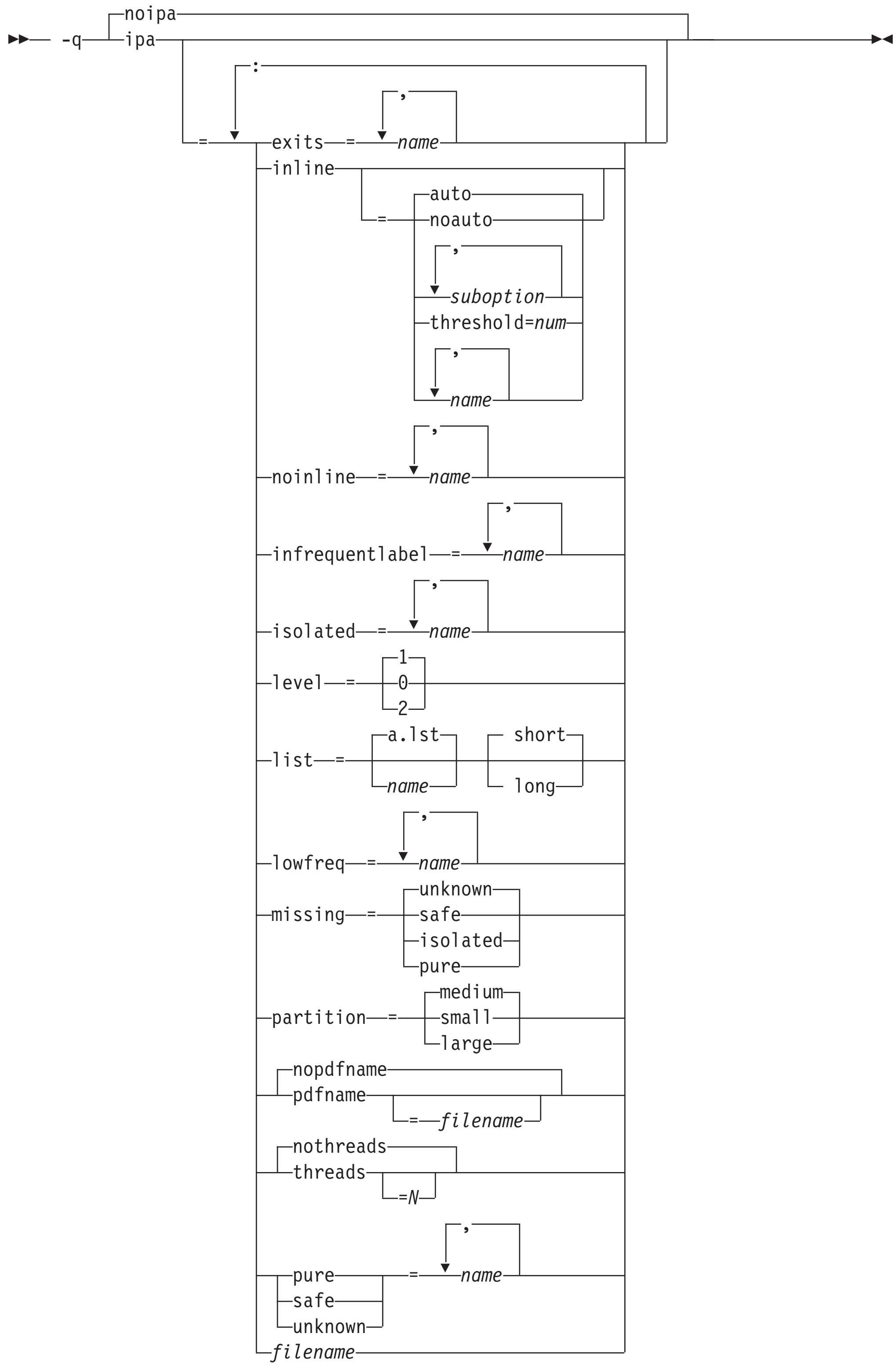
Compile-time syntax

```
►►── -qipa──┬─────────────────────┬──────────────────────────────────────►◄
            │      ┌─object───┐   │
            └─=────┴─noobject─┴───┘
```

where:

-qipa Compile-time Options	Description
-qipa	Activates interprocedural analysis with the following **-qipa** *suboption* defaults: • **inline=auto** • **level=1** • **missing=unknown** • **partition=medium**
-qipa=object -qipa=noobject	Specifies whether to include standard object code in the object files. Specifying the **noobject** suboption can substantially reduce overall compile time by not generating object code during the first IPA phase. If the **-S** compiler option is specified with **noobject**, **noobject** is ignored. If compilation and linking are performed in the same step, and neither the **-S** nor any listing option is specified, **-qipa=noobject** is implied by default. 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**.

Link-time syntax



where:

-qipa Link-time Options	Description
-qnoipa	Deactivates interprocedural analysis.

-qipa Link-time Options	Description
-qipa	Activates interprocedural analysis with the following **-qipa** *suboption* defaults: • **inline=auto** • **level=1** • **missing=unknown** • **partition=medium**

Suboptions can also include one or more of the forms shown below. Separate multiple suboptions with commas.

Link-time Suboptions	Description
exits=*name*{,*name*}	Specifies names of functions which represent program exits. Program exits are calls which can never return and can never call any procedure which has been compiled with IPA pass 1.
inline=auto inline=noauto	Enables or disables automatic inlining only. The compiler still accepts user-specified functions as candidates for inlining.
inline[=*suboption*]	Same as specifying the **-qinline** compiler option, with *suboption* being any valid **-qinline** suboption.
inline=threshold=*num*	Specifies an upper limit for the number of functions to be inlined, where *num* is a non-negative integer. This argument is implemented only when **inline=auto** is on.
inline=*name*{,*name*}	Specifies a comma-separated list of functions to try to inline, where functions are identified by *name*.
noinline=*name*{,*name*}	Specifies a comma-separated list of functions that must not be inlined, where functions are identified by *name*.
infrequentlabel=*name*{,*name*}	Specifies a list of user-defined labels that are likely to be called infrequently during a program run.
isolated=*name*,{*name*}	Specifies a list of *isolated* functions that are not compiled with IPA. Neither isolated functions nor functions within their call chain can refer to global variables.
level=0 level=1 level=2	Specifies the optimization level for interprocedural analysis. The default level is 1. Valid levels are as follows: • Level 0 - Does only minimal interprocedural analysis and optimization. • Level 1 - Turns on inlining, limited alias analysis, and limited call-site tailoring. • Level 2 - Performs full interprocedural data flow and alias analysis.

Link-time Suboptions	Description
list list=[*name*] [short|long]	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 generated by the back end for each partition. This option can also be used to specify the name of the listing file. If listings have been requested (using either the **-qlist** or **-qipa=list** options), and *name* is not specified, the listing file name defaults to **a.lst**. The **long** and **short** suboptions can be used to request more or less information in the listing file. The **short** suboption, which is the default, generates the Object File Map, Source File Map and Global Symbols Map sections of the listing. The **long** suboption causes the generation of all of the sections generated through the short suboption, as well as the Object Resolution Warnings, Object Reference Map, Inliner Report and Partition Map sections.
lowfreq=*name*{,*name*}	Specifies names of functions which are likely to be called infrequently. These will typically be error handling, trace, or initialization functions. The compiler may be able to make other parts of the program run faster by doing less optimization for calls to these functions.
missing=*attribute*	Specifies the interprocedural behavior of procedures that are not compiled with **-qipa** and are not explicitly named in an **unknown**, **safe**, **isolated**, or **pure** suboption. The following attributes may be used to refine this information: • safe - Functions which do not indirectly call a visible (not missing) function either through direct call or through a function pointer. • isolated - Functions which do not directly reference global variables accessible to visible functions. Functions bound from shared libraries are assumed to be *isolated*. • pure - Functions which are *safe* and *isolated* and which do not indirectly alter storage accessible to visible functions. *pure* functions also have no observable internal state. • unknown - The default setting. This option greatly restricts the amount of interprocedural optimization for calls to *unknown* functions. Specifies that the missing functions are not known to be *safe*, *isolated*, or *pure*.
partition=small partition=medium partition=large	Specifies the size of each program partition created by IPA during pass 2.

Link-time Suboptions	Description
nopdfname pdfname pdfname=*filename*	Specifies the name of the profile data file containing the **PDF** profiling information. If you do not specify *filename*, the default file name is **._pdf**. The profile is placed in the current working directory or in the directory named by the **PDFDIR** environment variable. This lets you do simultaneous runs of multiple executables using the same **PDFDIR**, which can be useful when tuning with **PDF** on dynamic libraries.
nothreads threads threads=*N*	Specifies the number of threads the compiler assigns to code generation. Specifying **nothreads** is equivalent to running one serial process. This is the default. Specifying **threads** allows the compiler to determine how many threads to use, depending on the number of processors available. Specifying **threads**=*N* instructs the program to use *N* threads. Though *N* can be any integer value in the range of 1 to MAXINT, *N* is effectively limited to the number of processors available on your system.
pure=*name*{,*name*}	Specifies a list of *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=*name*{,*name*}	Specifies a list of *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**.
unknown=*name*{,*name*}	Specifies a list of *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 and dummy arguments.

Link-time Suboptions	Description

filename	Gives the name of a file which contains suboption information in a special format.
	
	The file format is the following:
	`# ... comment`
	attribute`{, `*attribute*`} = `*name*`{, `*name*`}`
	`missing = `*attribute*`}, `*attribute*`}`
	`exits = `*name*`{, `*name*`}`
	`lowfreq = `*name*`{, `*name*`}`
	`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`
	`prof [ = `*file-name*` ]`
	`noprof`
	`partition = small | medium | large | `*unsigned_int*
	
	where *attribute* is one of:
	• exits
	• lowfreq
	• unknown
	• safe
	• isolated
	• pure

Notes

This option turns on or customizes a class of optimizations known as interprocedural analysis (IPA).

- IPA can significantly increase compilation time, even with the **-qipa=noobject** option, so using IPA should be limited to the final performance tuning stage of development.
- Specify the **-qipa** option on both the compile and link steps of the entire application, or as much of it as possible. You should at least compile the file containing **main**, or at least one of the entry points if compiling a library.
- While IPA's interprocedural optimizations can significantly improve performance of a program, they can also cause previously incorrect but functioning programs to fail. Listed below are some programming practices that can work by accident without aggressive optimization, but are exposed with IPA:
  1. Relying on the allocation order or location of automatic variables. For example, taking the address of an automatic variable and then later comparing it with the address of another local to determine the growth direction of a stack. The C language does not guarantee where an automatic variable is allocated, or it's position relative to other automatics. Do not compile such a function with IPA(and expect it to work).
  2. Accessing an either invalid pointer or beyond an array's bounds. IPA can reorganize global data structures. A wayward pointer which may have previously modified unused memory may now trample upon user allocated storage.
- Ensure you have sufficient resources to compile with IPA. IPA can generate significantly larger object files than traditional compilations. As a result, the temporary storage location used to hold these intermediate files (by convention

/tmp) is sometimes too small. If a large application is being compiled, consider redirecting temporary storage with the TMPDIR environment variable.
- Ensure there is enough swap space to run IPA (at least 200Mb for large programs). Otherwise the operating system might kill IPA with a signal 9 , which cannot be trapped, and IPA will be unable to clean up its temporary files.
- 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, nm, or dump outputs. Using IPA together with the **-g** compiler will usually result in non-steppable output.

Regular expression syntax can be used when specifying a *name* for the following suboptions.
- exits
- inline, noinline
- isolated
- lowfreq
- pure
- safe
- unknown

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.
[*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`.

The necessary steps to use IPA are:

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

Note: If a severe error occurs during compilation, **-qipa** returns RC=1 and terminates. Performance analysis also terminates.

Example

To compile a set of files with interprocedural analysis, enter:

```
xlc++ -c -O3 *.C -qipa
xlc++ -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 exits two functions, *trace_error* and *debug_dump*, which are rarely executed.

```
xlc++ -c -O3 *.C -qipa=noobject
xlc++ -c *.o -qipa=lowfreq=trace_error,debug_dump
```

Related References

"Compiler Command Line Options" on page 35
"inline" on page 125
"libansi" on page 161
"list" on page 163
"pdf1, pdf2" on page 187
"S" on page 208

isolated_call

C C++

Purpose

Specifies functions in the source file that have no side effects.

Syntax

```
                          ┌─:─────────────────┐
                          ▼                   │
►►── -q──isolated_call──=───function_name─────┴──────────────────────────────►◄
```

where:

function_name Is the name of a function that does not have side effects, except changing the value of a variable pointed to by a pointer or reference parameter, or does not rely on functions or processes that have side effects.

Side effects are any changes in the state of the runtime environment. Examples of such changes are accessing a volatile object, modifying an external object, modifying a file, or calling another function that does any of these things. Functions with no side effects cause no changes to external and static variables.

function_name can be a list of functions separated by colons (:).

See also “#pragma isolated_call” on page 287 and “#pragma options” on page 299.

Notes

Marking a function as isolated can improve the runtime performance of optimized code by indicating the following to the optimizer:

- external and static variables are not changed by the called function
- calls to the function with loop-invariant parameters may be moved out of loops
- multiple calls to the function with the same parameter may be merged into one call
- calls to the function may be discarded if the result value is not needed

The **#pragma options isolated_call** directive must be specified at the top of the file, before the first C or C++ statement. You can use the **#pragma isolated_call** directive at any point in your source file.

If a function is incorrectly identified as having no side effects, the resultant program behavior might be unexpected or produce incorrect results.

Example

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
```

Related References

“Compiler Command Line Options” on page 35
“#pragma isolated_call” on page 287
“#pragma options” on page 299

keepparm

▶ C ▶ C++

Purpose

Ensures that function parameters are stored on the stack even if the application is optimized.

Syntax

```
          ┌─nokeepparm─┐
▶▶── -q───┴─keepparm───┴──────────────────────────────────────────▶◀
```

Notes

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.

Specifying **-qkeepparm** ensures that the parameters are stored on the stack even when optimizing. This compiler option ensures that the values of incoming parameters are available to tools, such as debuggers, by preserving those values on the stack. However, doing so may negatively affect application performance.

Related references

- "O, optimize" on page 179

keyword

▶ C ▶ C++

Purpose

This option 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─┘
```

Notes

By default all the built-in keywords defined in the C and C++ language standards are reserved as keywords. 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.

This option can be used with all C++ built-in keywords.

This option can also be used with the following C built-in keywords:

- asm
- __complex__ (C99)
- __imag__ (C99)
- inline
- __real__ (C99)
- restrict
- typeof

Example

You can reinstate bool with the following invocation:

```
xlc++ -qkeyword=bool
```

Related References

"Compiler Command Line Options" on page 35

L

C C++

Purpose

At link time, searches the path directory for library files specified by the **-l***key* option.

Syntax

```
►►── -L──directory──────────────────────────────────────────────────►◄
```

Default

The default is to search only the standard directories.

Notes

If the LIBPATH environment variable is set, the compiler will search for libraries first in directory paths specified by LIBPATH, and then in directory paths specified by the **-L** compiler option.

If the **-L***directory* option is specified both in the configuration file and on the command line, search paths specified in the configuration file are the first to be searched at link time.

Paths specified with the **-L** compiler option are *not* searched at run time.

Example

To compile myprogram.c so that the directory **/usr/tmp/old** and all other directories specified by the **-l** option are searched for the library **libspfiles.a**, enter:

```
xlc myprogram.c -lspfiles -L/usr/tmp/old
```

Related References

"Compiler Command Line Options" on page 35
"l" on page 142
Appendix C, "Libraries in XL C/C++," on page 375

l

▶ C ▶ C++

Purpose

Searches the specified library file, lib*key*.so, and then lib*key*.a for dynamic linking, or just lib*key*.a for static linking.

Syntax

```
►►— -l—key——————————————————————————————————————————————————►◄
```

Default

The compiler default is to search only some of the compiler run-time 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.

Notes

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***directory* option.

The C and C++ runtime libraries are automatically added.

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.

Example

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 Tasks

"Specify Compiler Options in a Configuration File" on page 24

Related References

"Compiler Command Line Options" on page 35
"B" on page 60
"L" on page 141
Appendix C, "Libraries in XL C/C++," on page 375

langlvl

C C++

Purpose

Selects the language level and language options for the compilation.

Syntax

```
              ┌─:─────────┐
              ▼           │
►►── -q──langlvl──=───suboption─┴──────────────────────────────────────►◄
```

where values for *suboption* are described below in the **Notes** section.

See also "#pragma langlvl" on page 289 and "#pragma options" on page 299.

Default

The default language level varies according to the command you use to invoke the compiler:

Invocation	Default language level
xlC/xlc++	extended
xlc	extc89
cc	extended
c89	stdc89
c99	stdc99

Notes

C You can also use either of the following pragma directives to specify the language level in your C language source program:

```
#pragma options langlvl=suboption
#pragma langlvl(suboption)
```

The **pragma** directive must appear before any noncommentary lines in the source code.

C For C programs, you can use the following **-qlanglvl** suboptions for *suboption*:

`classic`	Allows the compilation of non-stdc89 programs, and conforms closely to the K&R level preprocessor.
`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 to the ANSI C89 standard, also known as ISO C90.
`stdc99`	Compilation conforms to the ISO C99 standard.
`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.

`[no]ucs`	Under language levels **stdc99** and **extc99**, the default is **-qlanglvl=ucs** This option controls whether Unicode characters are allowed in identifiers, string literals and character literals in program source code. The Unicode character set is supported by the C standard. This character set contains the full set of letters, digits and other characters used by a wide range of languages, including all North American and Western European languages. Unicode characters can be 16 or 32 bits. The ASCII one-byte characters are a subset of the Unicode character set. When this option is set to yes, you can insert Unicode characters in your source files either directly or using a notation that is similar to escape sequences. Because many Unicode characters cannot be displayed on the screen or entered from the keyboard, the latter approach is usually preferred. Notation forms for Unicode characters are \u*hhhh* for 16-bit characters, or \U*hhhhhhhh* for 32-bit characters, where *h* represents a hexadecimal digit. Short identifiers of characters are specified by ISO/IEC 10646.

The following **-qlanglvl** suboptions are accepted but ignored by the C compiler. Use **-qlanglvl=extended**, **-qlanglvl=extc99**, or **-qlanglvl=extc89** to enable the functions that these suboptions imply. For other values of **-qlanglvl**, 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.

C++ For C++ programs, you can specify one or more of the following **-qlanglvl** suboptions for *suboption*:

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

`[no]anonstruct`

This suboption controls whether anonymous structs and anonymous classes are allowed in your C++ source.

By default, the compiler allows anonymous structs. This is an extension to the C++ standard and gives behavior that is compatible with the C++ compilers provided by Microsoft® Visual C++.

Anonymous structs typically are 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 this suboption is set, you receive a warning if your code declares an anonymous struct and **-qinfo=por** is specified. When you build with **-qlanglvl=noanonstruct**, an anonymous struct is flagged as an error. Specify **noanonstruct** for compliance with standard C++.

`[no]anonunion`

This suboption controls what members are allowed in anonymous unions.

When this suboption is set to **anonunion**, anonymous unions can have members of all types that standard C++ allows in non-anonymous unions. For example, non-data members, such as structs, 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.

By default, the compiler allows non-data members in anonymous unions. This is an extension to standard C++ and gives behavior that is compatible with the C++ compilers provided by previous versions of VisualAge® C++ and predecessor products, and Microsoft Visual C++.

When this option is set to **anonunion**, you receive a warning if your code uses the extension, unless you suppress the arning message with the **-qsuppress** option.

Set **noanonunion** for compliance with standard C++.

`[no]ansifor`	This suboption controls whether scope rules defined in the C++ standard apply to names declared in for-init statements.
	
	By default, standard C++ rules are used. For example 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-init-statement, is visible only within the for statement. To correct the error, either declare i outside the loop or set ansiForStatementScopes to no.
	
	Set **noansifor** to allow old language behavior. You may need to do this for code that was developed with other products, such as the compilers provided by earlier versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.

`[no]ansisinit`	This option works in the same way as g++ -fuse-cxa-atexit and is required for fully standards-compliant handling of static destructors.

[no]c99__func__	This suboption instructs the compiler to recognize the C99 **__func__** identifier. The **__func__** identifier behaves as if there is an implicit declaration like:
	`static const char __func__[] =` *`function_name`*`;`
	where *function_name* is the name of the function in which the **__func__** identifier appears.
	
	The effect of the **__func__** identifier can be seen in the following code segment:
	`void this_function()`
	`{`
	` printf("__func__ appears in %s", __func__);`
	`}`
	which outputs the following when run:
	`__func__ appears in this_function`
	
	The **c99__func__** suboption is enabled by default when **-qlanglvl=extended** is in effect. It can be enabled for any language level by specifying **-qlanglvl=c99__func__**, or disabled by specifying **-qlanglvl=noc99__func__**.
	
	The __C99__FUNC__ macro is defined to be 1 when **c99__func__** is in effect, and is undefined otherwise.

[no]c99complex	This suboption instructs the compiler to recognize C99 complex data types and related keywords. **Note:** Support for complex data types may vary among different C++ compilers, creating potential portability issues. The compiler will issue a portability warning message if you specify this compiler option together with **-qinfo=por**.
[no]c99compoundliteral	This suboption instructs the compiler to support the C99 compound literal feature.
[no]c99hexfloat	This option enables support for C99-style hexadecimal floating constants in C++ applications. This suboption is on by default for **-qlanglvl=extended**. When it is in effect, the compiler defines the macro __C99_HEX_FLOAT_CONST .

[no]c99vla	When **c99vla** is in effect, the compiler will support the use of C99-type variable length arrays in your C++ applications. The macro __C99_VARIABLE_LENGTH_ARRAY is defined with a value of 1. **Note:** In C++ applications, storage allocated for use by variable length arrays is not released until the function they reside in completes execution.
[no]dependentbaselookup	The default is **-qlanglvl=dependentbaselookup**. This suboption provides the ability to specify compilation in conformance with Issue 213 of TC1 of the C++ Standard. The default setting retains the behavior of previous XL C++ compilers with regard to the name lookup for a template base class of dependent type: 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. For compliance with TC1, specify **-qlanglvl=nodependentbaselookup**.
[no]gnu_assert	GNU C portability option to enable or disable support for the following GNU C system identification assertions: • #assert • #unassert • #cpu • #machine • #system
[no]gnu_complex	This suboption instructs the compiler to recognize GNU complex data types and related keywords. **Note:** Support for complex data types may vary among different C++ compilers, creating potential portability issues. The compiler will issue a portability warning message if you specify this compiler option together with **-qinfo=por**.
[no]gnu_computedgoto	GNU C portability option to enable support for computed gotos. This suboption is enabled for **-qlanglvl=extended**, and defines the macro __IBM_COMPUTED_GOTO.
[no]gnu_explicitregvar	GNU C portability option to control whether the compiler accepts and ignores the specification of explicit registers for variables.

[no]gnu_externtemplate

This suboption enables or disables extern template instantiations.

The default setting is **gnu_externtemplate** when compiling to the **extended** language level.

If **gnu_externtemplate** is in effect, you can declare a template instantiation to be extern by adding the keyword **extern** in front of an explicit C++ template instantiation. The **extern** keyword must be the first keyword in the declaration, and there can be only one **extern** keyword.

This does not instantiate the class or function. For both classes and functions, the extern template instantiation will prevent instantiation of parts of the template, provided that instantiation has not already been triggered by code prior to the extern template instantiation, and it is not explicitly instantiated nor explicitly specialized.

For classes, static data members and member functions will not be instantiated, but a class itself will be instantiated if required to map the class. Any required compiler generated functions (for example, default copy constructor) will be instantiated. For functions, the prototype will be instantiated but the body of the template function will not.

See the following examples:

```
template < class T > class C {
  static int i;
  void f(T) { }
};

template < class U > int C<U>::i = 0;
extern template class C<int>; // extern explicit
                              // template
                              // instantiation
C<int> c;  // does not cause instantiation of
           // C<int>::i  or C<int>::f(int) in
           // this file but class is
           // instantiated for mapping
C<char> d; // normal instantiations

==========================

template < class C > C foo(C c) { return c; }

extern template int foo<int>(int); // extern explicit
                                   // template
                                   // instantiation
int i = foo(1);    // does not cause instantiation
                   // of body of foo<int>
```

[no]gnu_include_next
GNU C portability option to enable or disable support for the GNU C **#include_next** preprocessor directive.

[no]gnu_labelvalue
GNU C portability option to enable or disable support for labels as values. This suboption is on by default for **-qlanglvl=extended**, and defines the macro __IBM_LABEL_VALUE.

[no]gnu_locallabel
GNU C portability option to enable or disable support for locally-declared labels.

gnu_membernamereuse
GNU C portability option to enable reusing a template name in a member list as a typedef.

[no]gnu_suffixij	GNU C portability option to enable or disable support for GCC style complex numbers. If **gnu_suffixij** is specified, a complex number can be ended with suffix i/I or j/J.
[no]gnu_varargmacros	This option is similar to **-qlanglvl=varargmacros**. The main differences are:

- An optional variable argument identifier may precede the ellipsis, allowing that identifier to be used in place of the macro __VA_ARGS__ . Whitespace may appear between the identifier and the ellipsis.
- The variable argument can be omitted.
- If the token paste operator (**##**) appears between the comma and the variable argument, the preprocessor removes the dangling comma (**,**) if the variable argument is not provided.
- The macro __IBM_MACRO_WITH_VA_ARGS is defined to 1.

Example 1 - Simple substitution:

```
#define debug(format, args...) printf(format, args)

debug("Hello %s\n", "Chris");
```

preprocesses to:

```
printf("Hello %s\n", "Chris");
```

Example 2 - Omitting the variable argument:

```
#define debug(format, args...) printf(format, args)

debug("Hello\n");
```

preprocesses to the following, leaving a dangling comma:

```
printf("Hello\n",);
```

Example 3 - Using the token paste operator to remove a dangling comma when the variable argument is omitted:

```
#define debug(format, args...) printf(format, ## args)

debug("Hello\n");
```

preprocesses to:

```
printf("Hello\n");
```

[no]gnu_warning	GNU C portability option to enable or disable support for the GNU C **#warning** preprocessor directive.

[no]illptom

This suboption controls what expressions can be used to form pointers to members. The XL C++ compiler can accept some forms that are in common use but do not conform to the C++ Standard.

By default, the compiler allows these forms. This is an extension to standard C++ and gives behavior that is compatible with the C++ compilers provided by earlier versions of VisualAge C++, its predecessor products, and Microsoft Visual C++.

When this suboption is set to **illptom**, you receive warnings if your code uses the extension, unless you suppress the warning messages with the **-qsuppress** option.

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;
```

Set **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;
```

[no]implicitint

This suboption controls whether the compiler will accept missing or partially specified types as implicitly specifying int. This is no longer accepted in the standard but may exist in legacy code.

With the suboption set to **noimplicitint**, all types must be fully specified.

With the suboption set to **implicitint**, 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. Note that the effect is as if the int specifier were present. This means that the specifier const, by itself, would specify a constant integer.

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, _cdecl)

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.

By default, the compiler sets **-qlanglvl=implicitint**. This is an extension to the C++ standard and gives behavior that is compatible with the C++ compilers provided by earlier versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.

For example, the return type of function MyFunction is int because it was omitted in the following code:

```
MyFunction()
{
   return 0;
}
```

Set **-qlanglvl=noimplicitint** for compliance with standard C++. For example, the function declaration above must be modified to:

```
int MyFunction()
{
   return 0;
}
```

[no]offsetnonpod	This suboption 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. By default, the compiler allows offsetof to be used with non-POD classes. This is an extension to the C++ standard, and gives behavior that is compatible with the C++ compilers provided by VisualAge C++ for OS/2® 3.0, VisualAge for C++ for Windows®, Version 3.5, and Microsoft Visual C++ When this option is set, you receive a warning if your code uses the extension, unless you suppress the warning message with the **-qsuppress** option. Set **-qlanglvl=nooffsetnonpod** for compliance with standard C++. Set **-qlanglvl=offsetnonpod** if your code applies 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
[no]olddigraph	This option controls whether old-style digraphs are allowed in your C++ source. It applies only when **-qdigraph** is also set. By default, the compiler supports only the digraphs specified in the C++ standard. Set **-qlanglvl=olddigraph** if your code contains at least one of following digraphs:

Digraph	Resulting Character
%%	# (pound sign)
%%%%	## (double pound sign, used as the preprocessor macro concatenation operator)

Set **-qlanglvl=noolddigraph** for compatibility with standard C++ and the extended C++ language level supported by previous versions of VisualAge C++ and predecessor products.

[no]oldfriend

This option controls whether friend declarations that name classes without elaborated class names are treated as C++ errors.

By default, the compiler lets you declare a friend class without elaborating the name of the class with the keyword class. This is an extension to the C++ standard and gives behavior that is compatible with the C++ compilers provided by earlier versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.

For example, the statement below declares the class IFont to be a friend class and is valid when the **oldfriend** suboption is set specified.

```
friend IFont;
```

Set the **nooldfriend** suboption for compliance with standard C++. The example declaration above causes a warning unless you modify it to the statement as below, or suppress the warning message with **-qsuppress** option.

```
friend class IFont;
```

[no]oldtempacc

This suboption controls whether access to a copy constructor to create a temporary object is always checked, even if creation of the temporary object is avoided.

By default, the compiler suppresses the access checking. This is an extension to the C++ standard and gives behavior that is compatible with the C++ compilers provided by VisualAge C++ for OS/2® 3.0, VisualAge for C++ for Windows, Version 3.5, and Microsoft Visual C++.

When this suboption is set to yes, you receive a warning if your code uses the extension, unless you disable the warning message with the **-qsuppress** option.

Set **-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 thrown 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&).

[no]oldtmplalign	This suboption specifies the alignment rules implemented in versions of the compiler (xlC) prior to Version 5.0. These earlier versions of the xlC compiler ignore alignment rules specified for nested templates. By default, these alignment rules are not ignored in VisualAge C++ 4.0 or later. 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
};
```

[no]oldtmplspec

This suboption controls whether template specializations that do not conform to the C++ standard are allowed.

By default, the compiler allows these old specializations (**-qlanglvl=nooldtmplspec**). This is an extension to standard C++ and gives behavior that is compatible with the C++ compilers provided by VisualAge C++ for OS/2 3.0, VisualAge for C++ for Windows, Version 3.5, and Microsoft Visual C++.

When **-qlanglvl=oldtmplspec** is set, you receive a warning if your code uses the extension, unless you suppress the warning message with the **-qsuppress** option.

For example, you can explicitly specialize the template class ribbon for type char with the following lines:

```
template<class T> class ribbon { /*...*/};
class ribbon<char> { /*...*/};
```

Set **-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> { /*...*/};
```

[no]redefmac

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

[no]trailenum

This suboption controls whether trailing commas are allowed in enum declarations.

By default, the compiler allows one or more trailing commas at the end of the enumerator list. This is an extension to the C++ standard, and provides compatibility with Microsoft Visual C++. The following enum declaration uses this extension:

```
enum grain { wheat, barley, rye,, };
```

Set **-qlanglvl=notrailenum** for compliance with standard C++ or with the **stdc89** language level supported by previous versions of VisualAge C++ and predecessor products.

[no]typedefclass	This suboption provides backwards compatibility with previous versions of VisualAge C++ and predecessor products. The current C++ standard does not allow a typedef name to be specified where a class name is expected. This option relaxes that restriction. Set **-qlanglvl=typedefclass** to allow the use of typedef names in base specifiers and constructor initializer lists. By default, a typedef name cannot be specified where a class name is expected.
[no]ucs	This suboption controls whether Unicode characters are allowed in identifiers, string literals and character literals in C++ sources. The default setting is **-qlanglvl=noucs**. The Unicode character set is supported by the C++ standard. This character set contains the full set of letters, digits and other characters used by a wide range of languages, including all North American and Western European languages. Unicode characters can be 16 or 32 bits. The ASCII one-byte characters are a subset of the Unicode character set. When **-qlanglvl=ucs** is set, you can insert Unicode characters in your source files either directly or using a notation that is similar to escape sequences. Because many Unicode characters cannot be displayed on the screen or entered from the keyboard, the latter approach is usually preferred. Notation forms for Unicode characters are \u*hhhh* for 16-bit characters, or \U*hhhhhhhh* for 32-bit characters, where *h* represents a hexadecimal digit. Short identifiers of characters are specified by ISO/IEC 10646.

[no]varargmacros	This C99 feature allows the use of a variable argument list in function-like macros in your C++ applications. The syntax is similar to a variable argument function, and can be used as a masking macro for printf.
	
	For example:
	`#define debug(format, ...) printf(format, __VA_ARGS__)`
	
	`debug("Hello %s\n", "Chris");`
	
	preprocesses to::
	`printf("Hello %s\n", "Chris");`
	
	The token __VA_ARGS__ in the replacement list corresponds to the ellipsis in the parameter. The ellipsis represents the variable arguments in a macro invocation.
	
	Specifying **varargmacros** defines the macro __C99_MACRO_WITH_VA_ARGS to a value of 1.

[no]zeroextarray

This suboption controls whether zero-extent arrays are allowed as the last non-static data member in a class definition.

By default, the compiler allows arrays with zero elements. This is an extension to the C++ standard, and provides compatibility with Microsoft Visual C++. The example declarations below define dimensionless arrays a and b.

```
struct S1 { char a[0]; };
struct S2 { char b[]; };
```

Set **nozeroextarray** for compliance with standard C++ or with the ANSI language level supported by previous versions of VisualAge C++ and predecessor products.

Exceptions to the **stdc89** mode addressed by **classic** are as follows:

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 between **classic** mode and **stdc89** mode. Directives not listed here behave similarly in both modes.

#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 in **classic** mode.

#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.

Related references

- “bitfields” on page 62
- “chars” on page 68
- “digraph” on page 84
- “flag” on page 98
- “info” on page 119
- “inline” on page 125
- “M” on page 168
- “ro” on page 205
- “suppress” on page 227
- “#pragma langlvl” on page 289
- “#pragma options” on page 299

- See also ″IBM C Language Extensions″ and ″IBM C++ Language Extensions″ in *XL C/C++ Language Reference*.

lib

C C++

Purpose

Instructs the compiler to use the standard system libraries at link time.

Syntax

```
          ┌─lib───┐
►►── -q───┴─nolib─┴────────────────────────────────────────────────►◄
```

Notes

If the **-qnolib** compiler option is specified, the standard system libraries are not used. Only those libraries explicitly specified on the command line will be used at link time.

Related References

"Compiler Command Line Options" on page 35
"crt" on page 79

libansi

▶ C ▶ C++

Purpose

Assumes that all functions with the name of an ANSI C library function are in fact the system functions.

Syntax

```
                  ┌─nolibansi─┐
▶▶── -q───────────┴─libansi───┴──────────────────────────────────────────▶◀
```

See also “#pragma options” on page 299.

Notes

This will allow the optimizer to generate better code because it will know about the behavior of a given function, such as whether or not it has any side effects.

Related References

“Compiler Command Line Options” on page 35
“#pragma options” on page 299

linedebug

C C++

Purpose

Generates line number and source file name information for the debugger.

Syntax

```
          ┌─nolinedebug─┐
►►── -q───┴─linedebug───┴──────────────────────────────────────────────►◄
```

Notes

This option produces minimal debugging information, so the resulting object size is smaller than that produced if the **-g** debugging option is specified. 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.

Avoid using this option with **-O** (optimization) option. The information produced may be incomplete or misleading.

If you specify the **-qlinedebug** option, the inlining option defaults to **-Q!** (no functions are inlined).

The **-g** option overrides the **-qlinedebug** option. If you specify **-g -qnolinedebug** on the command line, **-qnolinedebug** is ignored and the following warning is issued:

```
1506-... (W) Option -qnolinedebug is incompatible with option -g and is ignored
```

Example

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 References

"Compiler Command Line Options" on page 35
"g" on page 107
"O, optimize" on page 179
"Q" on page 199
"#pragma options" on page 299

list

C C++

Purpose

Produces a compiler listing that includes an object listing.

Syntax

```
          ┌nolist┐
►►── -q───┴list──┴──────────────────────────────────────────────────►◄
```

See also “#pragma options” on page 299.

Notes

The **-qnoprint** compiler option overrides this option.

Example

To compile myprogram.C and produce an object listing, enter:

```
xlc++ myprogram.C -qlist
```

Related References

“Compiler Command Line Options” on page 35
“print” on page 194
“#pragma options” on page 299

listopt

C C++

Purpose

Produces a compiler listing that displays all options in effect at time of compiler invocation.

Syntax

```
         ┌─nolistopt─┐
▶▶── -q──┴─listopt───┴──────────────────────────────────────────────▶◀
```

Notes

The listing will show options in effect as set by the compiler defaults, default configuration file, and command line settings. Option settings caused by **#pragma** statements in the program source are not shown in the compiler listing.

Specifying **-qnoprint** overrides this compiler option.

Example

To compile myprogram.C to produce a compiler listing that shows all options in effect, enter:

```
xlc++ myprogram.C -qlistopt
```

Related References

"Compiler Command Line Options" on page 35
"print" on page 194
"Resolving Conflicting Compiler Options" on page 27

longlit

C C++

Purpose

Makes unsuffixed literals into the long type in 64-bit mode.

Syntax

```
           ┌─nolonglit─┐
►►── -q────┴─longlit───┴──────────────────────────────────────────►◄
```

Notes

The following table shows the implicit types for constants in 64-bit mode when compiling in the **stdc89**, **extc89**, or **extended** language level:

	default 64-bit mode	**64-bit mode with qlonglit**

unsuffixed decimal	`signed int`
	`signed long`
	`unsigned long`
		`signed long`
		`unsigned long`

unsuffixed octal or hex	`signed int`
	`unsigned int`
	`signed long`
	`unsigned long`
		`signed long`
		`unsigned long`

suffixed by u/U	`unsigned int` `unsigned long`	`unsigned long`
suffixed by l/L	`signed long` `unsigned long`	`signed long` `unsigned long`
suffixed by ul/UL	`unsigned long`	`unsigned long`

The following table shows the implicit types for constants in 64-bit mode when compiling in the **stdc99**, **extc99**, or **extended** language level:

	Decimal Constant	**-qlonglit effect on Decimal Constant**
unsuffixed	`int` `long int`	`long int`
u or **U**	`unsigned int` `unsigned long int`	`unsigned long int`
l or **L**	`long int`	`long int`
Both **u** or **U**, and **l** or **L**	`unsigned long int`	`unsigned long int`
ll or **LL**	`long long int`	`long long int`
Both **u** or **U**, and **ll** or **LL**	`unsigned long long int`	`unsigned long long int`

	Octal or Hexadecimal Constant	**-qlonglit effect on Octal or Hexadecimal Constant**

unsuffixed	`int`
	`unsigned int`
	`long int`
	`unsigned long int`
		`long int`
		`unsigned long int`

	Octal or Hexadecimal Constant	-qlonglit effect on Octal or Hexadecimal Constant
u or **U**	`unsigned int` `unsigned long int`	`unsigned long int`
l or **L**	`long int` `unsigned long int`	`long int` `unsigned long int`
Both **u** or **U**, and **l** or **L**	`unsigned long int`	`unsigned long int`
ll or **LL**	`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`

Related References

"Compiler Command Line Options" on page 35
"langlvl" on page 143

longlong

C C++

Purpose

Allows **long long** integer types in your program.

Syntax

```
          ┌─longlong───┐
►►── -q───┴─nolonglong─┴──────────────────────────────────────────►◄
```

Default

The default with **xlc** , **xlC** and **cc** is **-qlonglong**, which defines **_LONG_LONG** (**long long** types will work in programs). The default with **c89** is **-qnolonglong** (**long long** types are not supported).

Notes

C This option cannot be specified when the selected language level is **stdc99** or **extc99**. It is used to control the long long support that is provided as an extension to the C89 standard. This extension is slightly different from the long long support that is part of the C99 standard.

Example

1. To compile myprogram.c so that **long long int**s are not allowed, enter:

```
xlc myprogram.c -qnolonglong
```

Related References

"Compiler Command Line Options" on page 35

M

C C++

Purpose

Creates an output file that contains targets suitable for inclusion in a description file for the **make** command.

Syntax

```
►►── -M──────────────────────────────────────────────────────────────►◄
```

Notes

The **-M** option is functionally identical to the **-qmakedep** option.

.d files are not **make** files; **.d** files must be edited 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:file_name.c
file_name.o:include_file_name
```

Include files are listed according to the search order rules for the **#include** preprocessor directive, described in *Directory Search Sequence for Include Files Using Relative Path Names*. If the include file is not found, it is not added to the **.d** file.

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

Examples

If you do not specify the **-o** option, the output file generated by the **-M** option is created in the current directory. It has a **.d** suffix. For example, the command:

```
xlc -M person_years.c
```

produces the output file **person_years.d**.

A **.d** file is created for every input file with a **.c**, **.C**, **.cpp**, or **.i** suffix. Also, when compiling C++ programs with the **-+** compiler option in effect, any file suffix is accepted and a **.d** file produced. Otherwise, output **.d** files are not created for any other files.

For example, the command:

```
xlc -M conversion.c filter.c /lib/libm.a
```

produces two output files, **conversion.d** and **filter.d**, and an executable file as well. No **.d** file is created for the library.

If the current directory is not writable, no **.d** file is created. If you specify **-o** *file_name* along with **-M**, the **.d** file is placed in the directory implied by **-o** *file_name*. For example, for the following invocation:

```
xlc -M -c t.c -o /tmp/t.o
```

places the **.d** output file in **/tmp/t.d**.

Related Tasks

"Directory Search Sequence for Include Files Using Relative Path Names" on page 29

Related References

"Compiler Command Line Options" on page 35
"+ (plus sign)" on page 43
"makedep" on page 171
"o" on page 183
"sourcetype" on page 218

ma

C

Purpose

Substitutes inline code for calls to built-in function **alloca**.

Syntax

```
►►── -ma──────────────────────────────────────────────────────────────►◄
```

Notes

If **#pragma alloca** is unspecified, or if you do not use **-ma**, **alloca** is treated as a user-defined identifier rather than as a built-in function.

This option does not apply to C++ programs. In C++ programs, you must instead specify **#include <malloc.h>** to include the **alloca** function declaration.

Example

To compile myprogram.c so that calls to the function **alloca** are treated as inline, enter:

```
xlc myprogram.c -ma
```

Related References

"Compiler Command Line Options" on page 35
"alloca" on page 54
"#pragma alloca" on page 262

makedep

C C++

Purpose

Creates an output file that contains targets suitable for inclusion in a description file for the **make** command.

Syntax

```
►►── -q──makedep──────────────────────────────────────────────────────►◄
```

Notes

The **-qmakedep** option is functionally identical to the **-M** option.

.d files are not **make** files; **.d** files must be edited before they can be used with the **make** command. For more information on this command, see your operating system documentation.

If you do not specify the **-o** option, the output file generated by the **-qmakedep** option is created in the current directory. It has a **.d** suffix. For example, the command:

```
xlc++ -qmakedep person_years.C
```

produces the output file **person_years.d**.

A **.d** file is created for every input file with a **.c**, **.C**, **.cpp**, or **.i** suffix. Also, when compiling C++ programs with the **-+** compiler option in effect, any file suffix is accepted and a **.d** file produced. Otherwise, output **.d** files are not created for any other files.

For example, the command:

```
xlc++ -qmakedep conversion.C filter.C /lib/libm.a
```

produces two output files, **conversion.d** and **filter.d** (and an executable file as well). No **.d** file is created for the library.

If the current directory is not writable, no **.d** file is created. If you specify **-o** *file_name* along with **-qmakedep**, the **.d** file is placed in the directory implied by **-o***file_name*. For example, for the following invocation:

```
xlc++ -qmakedep -c t.C -o /tmp/t.o
```

places the **.d** output file in **/tmp/t.d**.

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.C
```

Include files are listed according to the search order rules for the **#include** preprocessor directive, described in "Directory Search Sequence for Include Files Using Relative Path Names" on page 29. If the include file is not found, it is not added to the **.d** file.

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

Related References

"Compiler Command Line Options" on page 35
"M" on page 168
"o" on page 183
"Directory Search Sequence for Include Files Using Relative Path Names" on page 29

maxerr

C C++

Purpose

Instructs the compiler to halt compilation when *num* errors of a specified severity level or higher is reached.

Syntax

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

where *num* must be an integer. Choices for severity level can be one of the following:

sev_level	**Description**
i	Informational
w	Warning
e	Error (C only)
s	Severe error

Notes

If a severity level is not specified, the current value of the **-qhalt** option is used.

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.

An unrecoverable error occurs when the number of errors reached the limit specified. The error message issued is similar to:

```
1506-672 (U) The number of errors has reached the limit of ...
```

If **-qnomaxerr** is specified, the entire source file is compiled regardless of how many errors are encountered.

Diagnostic messages may be controlled by the **-qflag** option.

Examples

1. To stop compilation of myprogram.c when 10 warnings are encountered, enter the command:
   ```
   xlc myprogram.c -qmaxerr=10:w
   ```
2. 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
   ```
3. 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 References

"Compiler Command Line Options" on page 35
"flag" on page 98
"halt" on page 111
"Message Severity Levels and Compiler Response" on page 345

maxmem

C C++

Purpose

Limits the amount of memory used by the optimizer for local tables of specific, memory-intensive optimizations. The memory size limit is specified in kilobytes.

Syntax

```
►►── -q──maxmem──=──size──────────────────────────────────────────►◄
```

Defaults

- With -O2 optimization in effect, maxmem=8192.
- With -O3 or greater optimization in effect, maxmem=-1.

Notes

- A *size* value of -1 permits each optimization to take as much memory as it needs without checking for limits. Depending on the source file being compiled, the size of subprograms in the source, the machine configuration, and the workload on the system, this might exceed available system resources.
- The limit set by **-qmaxmem** 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.
- Setting a large limit has no negative effect on the compilation of source files when the compiler needs less memory.
- Limiting the scope of optimization 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.

Depending on the source file being compiled, the size of the subprograms in the source, the machine configuration, and the workload on the system, setting the limit too high might lead to page-space exhaustion. In particular, specifying **-qmaxmem=-1** allows the compiler to try and use an infinite amount of storage, which in the worst case can exhaust the resources of even the most well-equipped machine.

Example

To compile myprogram.C so that the memory specified for local table is **16384** kilobytes, enter:

```
xlc++ myprogram.C -qmaxmem=16384
```

Related References

"Compiler Command Line Options" on page 35

mbcs, dbcs

C C++

Purpose

Use the **-qmbcs** option if your program contains multibyte characters. The **-qmbcs** option is equivalent to **-qdbcs**.

Syntax

```
          ┌─nombcs─┐
>>── -q───┼─mbcs───┼──────────────────────────────────────────────><
          ├─nodbcs─┤
          └─dbcs───┘
```

See also "#pragma options" on page 299.

Notes

Multibyte characters are used in certain languages such as Chinese, Japanese, and Korean.

Multibyte characters are also permitted in comments, if you specify the **-qmbcs** or **-qdbcs** compiler option.

If a source file contains multibyte character literals and the default **-qnombcs** or **-qnodbcs** compiler option is in effect, the compiler will treat all literals as single-byte literals.

Example

To compile myprogram.c if it contains multibyte characters, enter:

```
xlc myprogram.c -qmbcs
```

Related References

"Compiler Command Line Options" on page 35
"#pragma options" on page 299

minimaltoc

C C++

Purpose

Avoids toc overflow conditions in 64-bit compilations by placing toc entries into a separate data section for each object file.

Syntax

```
          ┌─nominimaltoc─┐
►►── -q───┴─minimaltoc───┴──────────────────────────────────────────────►◄
```

Notes

This compiler option applies to 64-bit compilations only.

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. You can avoid such toc overflow errors by compiling with the **-qminimaltoc** option.

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.

Related References

“Compiler Command Line Options” on page 35
“O, optimize” on page 179

mkshrobj

C C++

Purpose

Creates a shared object from generated object files.

Syntax

```
>>— -q—mkshrobj————————————————————————————————————————————————><
```

Notes

This option, together with the related options described below, is used to create a shared object. The advantage to using this option is that the compiler will automatically include and compile the template instantiations in the tempinc directory.

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

Also, the following related option can be used with the **-qmkshrobj** compiler option:

-o *shared_file*	Is the name of the file that will hold the shared file information. The default is `a.out`.

If you use **-qmkshrobj** to create a shared library, the compiler and linker are called with the appropriate options to build a shared object.

Example

To construct the shared library **big_lib.o** from three smaller object files, type:

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

Related References

"Compiler Command Line Options" on page 35
"32, 64" on page 45
"e" on page 90
"lib" on page 160
"o" on page 183
"path" on page 186
"pic" on page 192
"priority" on page 195
"#pragma priority" on page 308

See also the *Creating a Library* section of the *XL C/C++ Programming Guide*.

O, optimize

C C++

Purpose

Specifies whether to optimize code during compilation, and if so, specifies the optimization level.

Syntax

```
           ┌nooptimize──────────────┐
►►──┬─ -q──┴optimize───┬──────────┬─┴──┬──────────────────────────────────►◄
    │                  └=──┬─0─┬──┘    │
    │                      ├─2─┤       │
    │                      ├─3─┤       │
    │                      ├─4─┤       │
    │                      └─5─┘       │
    ├─ -O0─────────────────────────────┤
    ├─ -O──────────────────────────────┤
    ├─ -O2─────────────────────────────┤
    ├─ -O3─────────────────────────────┤
    ├─ -O4─────────────────────────────┤
    └─ -O5─────────────────────────────┘
```

where optimization settings are:

-O0 -qNOOPTimize -qOPTimize=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 -qOPTimize	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**.
-O2 -qOPTimize=2	Same as **-O**.
-O3 -qOPTimize=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**. Use the **-qstrict** option with **-O3** to turn off the aggressive optimizations 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. The **-qstrict** compiler option must appear after the **-O3** option, otherwise it is ignored.

<table>
<tr>
<td>-O3
-qOPTimize=3
(continued)</td>
<td>

The aggressive optimizations performed when you specify **-O3** are:

1. Aggressive code motion, and scheduling on computations that have the potential to raise an exception, are allowed.

   Loads and floating-point computations fall into this category. This optimization is aggressive because it may place such instructions onto execution paths where they *will* be executed when they *may* not have been according to the actual semantics of the program.

   For example, a loop-invariant floating-point computation that is found on some, but not all, paths through a loop will not be moved at **-O2** because the computation may cause an exception. At **-O3**, the compiler will move it because it is not certain to cause an exception. The same is true for motion of loads. Although a load through a pointer is never moved, loads off the static or stack base register are considered movable at **-O3**. Loads in general are not considered to be absolutely safe at **-O2** because a program can contain a declaration of a static array `a` of 10 elements and load `a[60000000003]`, which could cause a segmentation violation.

   The same concepts apply to scheduling.

   Example:

   In the following example, at **-O2**, the computation of `b+c` is not moved out of the loop for two reasons:

   - It is considered dangerous because it is a floating-point operation
   - t does not occur on every path through the loop

   At **-O3**, the code is moved.

   ```
     ...
   int i ;
   float a[100], b, c ;
   for (i = 0 ; i < 100 ; i++)
    {
    if (a[i] < a[i+1])
     a[i] = b + c ;
    }
     ...
   ```

2. Conformance to IEEE rules are relaxed.

   With **-O2** certain optimizations are not performed because they may produce an incorrect sign in cases with a zero result, and because they remove an arithmetic operation that may cause some type of floating-point exception.

   For example, **X + 0.0** is not folded to X because, under IEEE rules, **-0.0 + 0.0 = 0.0**, which is **-X**. In some other cases, some optimizations may perform optimizations that yield a zero result with the wrong sign. For example, **X - Y * Z** may result in a **-0.0** where the original computation would produce **0.0**.

   In most cases the difference in the results is not important to an application and **-O3** allows these optimizations.

3. Floating-point expressions may be rewritten.

   Computations such as **a*b*c** may be rewritten as **a*c*b** if, for example, an opportunity exists to get a common subexpression by such rearrangement. Replacing a divide with a multiply by the reciprocal is another example of reassociating floating-point computations.

</td>
</tr>
</table>

-O3, -qOPTimize=3 (*continued*)	Notes • **-qfloat=rsqrt** is set by default with **-O3**. • **-qmaxmem=1** is set by default with **-O3**, allowing the compiler to use as much memory as necessary when performing optimizations. • Built-in functions do not change **errno** at **-O3**. • Integer divide instructions are considered too dangerous to optimize even at **-O3**. • The default **-qmaxmem** value is **-1** at **-O3**. • Refer to **-qflttrap** to see the behavior of the compiler when you specify **optimize** options with the **flttrap** 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. 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.
-O4 -qOPTimize=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 -qOPTimize=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.

Notes

You can abbreviate **-qoptimize...** to **-qopt...**. For example, **-qnoopt** is equivalent to **-qnooptimize**.

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.

Example

To compile myprogram.C for maximum optimization, enter:

```
xlc++ myprogram.C -O3
```

Related references

- See also: ″Getting started with optimization″ in *Getting Started with XL C/C++* and ″Optimizing your applications″ in *XL C/C++ Programming Guide*.

o

C C++

Purpose

Specifies an output location for the object, assembler, or executable files created by the compiler. When the **-o** option is used during compiler invocation, *filespec* can be the name of either a file or a directory. When the **-o** option is used during direct linkage-editor invocation, *filespec* can only be the name of a file.

Syntax

```
▸▸— -o— filespec——————————————————————————————————————————▸◂
```

Notes

When **-o** is specified as part of a compiler invocation, *filespec* can be the relative or absolute path name of either a directory or a file.

1. If *filespec* is the name of a directory, files created by the compiler are placed into that directory.
2. If a directory with the name *filespec* does not exist, the **-o** option specifies that the name of the file produced by the compiler will be *filespec*. For example, the compiler invocation:

   ```
   xlc test.c -c -o new.o
   ```

   produces the object file **new.o** instead of **test.o** , and

   ```
   xlc test.c -o new
   ```

   produces the object file **new** instead of **a.out**, provided there is no directory also named **new**. Otherwise, the default object name **a.out** is used and placed in the **new** directory.

   A *filespec* with a C or C++ source file suffix (**.C**, **.c**, **.cpp**, or **.i**), such as myprog.c or myprog.i, results in an error and neither the compiler nor the linkage editor is invoked.

   If you use **-c** and **-o** together and the *filespec* does not specify a 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***filename* option.

Example

To compile myprogram.c so that the resulting file is called **myaccount**, assuming that no directory with name **myaccount** exists, enter:

```
xlc myprogram.c -o myaccount
```

If the directory **myaccount** does exist, the compiler produces the executable file **a.out** and places it in the **myaccount** directory.

Related References

"Compiler Command Line Options" on page 35
"c" on page 64
"E" on page 88
"P" on page 184
"syntaxonly" on page 229

P

▶ C ▶ C++

Purpose

Preprocesses the C or C++ source files named in the compiler invocation and creates an output preprocessed source file, *file_name*.i for each input source file *file_name*.c, *file_name*.C, or *file_name*.cpp.

Syntax

```
►►── -P ─────────────────────────────────────────────────────────►◄
```

Notes

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.

#line directives are not issued.

The **-P** option cannot accept a preprocessed source file, such as *file_name***.i** as input. The compiler will issue an error message.

Source files with unrecognized filename suffixes are treated and preprocessed as C files, and no error message is generated.

In extended mode, the preprocessor interprets the backslash character when it is followed by a new-line character as line-continuation in:

- macro replacement text
- macro arguments
- comments that are on the same line as a preprocessor directive.

Line continuations elsewhere are processed in **ANSI** mode only.

The **-P** option is overridden by the **-E** option. The **-P** option overrides the **-c**, **-o,** and **-qsyntaxonly** option. The **-C** option may used in conjunction with both the **-E** and **-P** options.

The default is to compile and link-edit C or C++ source files to produce an executable file.

Related References

"Compiler Command Line Options" on page 35
"C" on page 63
"c" on page 64
"E" on page 88
"o" on page 183
"syntaxonly" on page 229

p

C C++

Purpose

Sets up the object files produced by the compiler for profiling.

Syntax

```
►►── -p──────────────────────────────────────────────────────────────►◄
```

Notes

If the **-qtbtable** option is not set, the **-p** option will generate full traceback tables.

When compiling and linking in separate steps, the **-p** option must be specified in both steps.

Example

To compile myprogram.c so that it can be used with your operating system's **gprof** command, enter:

```
xlc++ myprogram.C -p
```

Related References

"Compiler Command Line Options" on page 35
"tbtable" on page 232

path

C C++

Purpose

Constructs alternate program names for compiler components. The program and directory *path* specified by this option is used in place of the regular program.

Syntax

```
>>── -q──path──=──┬─c─┬──:──path──────────────────────────────────────><
                  ├─b─┤
                  ├─p─┤
                  ├─a─┤
                  ├─I─┤
                  ├─L─┤
                  └─l─┘
```

where program names are:

Program	Description
c	Compiler front end
b	Compiler back end
p	Compiler preprocessor
a	Assembler
I	Interprocedural Analysis tool - compile phase
L	Interprocedural Analysis tool - link phase
l	Linkage editor

Notes

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

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 linkage editor in **/lib/tmp/mine/**, enter:

```
xlc++ myprogram.C -qpath=l:/lib/tmp/mine/
```

Related References

"Compiler Command Line Options" on page 35
"B" on page 60
"F" on page 97
"t" on page 230

pdf1, pdf2

C C++

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.

Syntax

```
          ┌─nopdf2─┐
          ├─nopdf1─┤
►►── -q ──┼─pdf1───┼──────────────────────────────────────────────────►◄
          └─pdf2───┘
```

Notes

To use PDF, follow these steps:

1. Compile some or all of the source files in a program with the **-qpdf1** option. You need to specify at least the **-O2** optimizing option and you also need to link with at least **-O2** in effect as well. Pay special attention to the compiler options that you use to compile the files, because you will need to use the same options later.

   In a large application, concentrate on those areas of the code that can benefit most from optimization. You do not need to compile all of the application's code with the **-qpdf1** option.
2. Run the program all the way through using a typical data set. The program records profiling information when it finishes. You can run the program multiple times with different data sets, and the profiling information is accumulated to provide an accurate count of how often branches are taken and blocks of code are executed.

   Important: Use data that is representative of the data that will be used during a normal run of your finished program.
3. Relink your program using the same compiler options as before, but change **-qpdf1** to **-qpdf2**. Remember that **-L**, **-l**, and some others are linker options, and you can change them at this point. In this second compilation, the accumulated profiling information is used to fine-tune the optimizations. The resulting program contains no profiling overhead and runs at full speed.

For best performance, use the **-O3**, **-O4**, or **-O5** option with all compilations when you use PDF.

The profile is placed in the current working directory or in the directory that the PDFDIR environment variable names, if that variable is set.

To avoid wasting compilation and execution time, make sure that the PDFDIR environment variable is set to an absolute path. Otherwise, you might run the application from the wrong directory, and it will not be able to locate the profile data files. When that happens, the program may not be optimized correctly or may be stopped by a segmentation fault. A segmentation fault might also happen if you change the value of the PDFDIR variable and execute the application before finishing the PDF process.

Because this option requires compiling the entire application twice, it is intended to be used after other debugging and tuning is finished, as one of the last steps before putting the application into production.

Restrictions

- PDF optimizations require at least the **-O2** optimization level.
- You must compile the main program with PDF for profiling information to be collected at run time.
- Do not compile or run two different applications that use the same PDFDIR directory at the same time, unless you have used the **-qipa=pdfname** suboption to distinguish the sets of profiling information.
- You must use the same set of compiler options at all compilation steps for a particular program. Otherwise, PDF cannot optimize your program correctly and may even slow it down. All compiler settings must be the same, including any supplied by configuration files.
- Avoid mixing PDF files created by the current version level of XL C/C++ with PDF files created by other version levels of the compiler.
- If **-qipa** is not invoked either directly or through other options, **-qpdf1** and **-qpdf2** will invoke the **-qipa=level=0** option.
- If you compile a program with **-qpdf1**, remember that it will generate profiling information when it runs, which involves some performance overhead. This overhead goes away when you recompile with **-qpdf2** or with no PDF at all.

The following utility programs, found in **/usr/xlopt/bin**, are available for managing the **PDFDIR** directory:

cleanpdf `cleanpdf [`*pathname*`]`

Removes all profiling information from the *pathname* directory; or if *pathname* is not specified, from the **PDFDIR** directory; or if **PDFDIR** is not set, from the current directory. Removing profiling information reduces run-time 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 [-r `*scaling*`] input {[-r `*scaling*`] input} ... –o `*output*` [-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.

record
: 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 the screen.

resetpdf `resetpdf [pathname]`

Same as `cleanpdf [pathname]`, described above.

showpdf `showpdf`

Displays the call and block counts for all procedures executed in a program run. To use this command, you must first compile your application specifying both **-qpdf1** and **-qshowpdf** compiler options on the command line.

Examples

Here is a simple example:

```
/* Set the PDFDIR variable.                        */
export PDFDIR=$HOME/project_dir

/* 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 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++ file1.o file2.o file3.o file4.o
```

Related References

"Compiler Command Line Options" on page 35
"ipa" on page 129
"O, optimize" on page 179
"showpdf" on page 212

pg

▶ C ▶ C++

Purpose

Sets up the object files for profiling.

If the **-qtbtable** option is not set, the **-pg** option will generate full traceback tables.

Syntax

```
▶▶── -pg──────────────────────────────────────────────────────────▶◀
```

Example

To compile myprogram.c for use with your operating system's **gprof** command, enter:

```
xlc myprogram.c -pg
```

Remember to compile *and* link with the **-pg** option. For example:

```
xlc myprogram.c -pg -c
xlc myprogram.o -pg -o program
```

Related References

"Compiler Command Line Options" on page 35
"tbtable" on page 232

phsinfo

C C++

Purpose

Reports the time taken in each compilation phase. Phase information is sent to standard output.

Syntax

```
            ┌─nophsinfo─┐
►►── -q─────┴─phsinfo───┴──────────────────────────────────────────►◄
```

Notes

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.

Example

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
```

Related References

"Compiler Command Line Options" on page 35

pic

C C++

Purpose

Instructs the compiler to generate Position-Independent Code suitable for use in shared libraries.

Syntax

```
              ┌─nopic──────────────────┐
              │          ┌─=──small─┐  │
►►── -q──────┴─pic──────┤          ├──┴──────────────────────────────────────────►◄
                         └─=──large─┘
```

where

nopic
: Instructs the compiler to not generate Position Independant Code.

pic
: Instructs the compiler to generate Position Independant Code.

small
: Instructs the compiler to assume that the size of the Global Offset Table is no larger than 64 Kb.

large
: Allows the Global Offset Table to be larger than 64 Kb in size, allowing more addresses to be stored in the table. Code generated with this option is usually larger than that generated with **-qpic=small**.

Notes

If **-qpic** is specified without any suboptions, **-qpic=small** is assumed.

The **-qpic** option is implied if the **-qmkshrobj** compiler option is specified.

Specifying **-q64** automatically implies **-qpic**.

Example

To compile a shared library **libmylib.so**, use the following command:

```
xlc mylib.c -qpic -Wl, -shared, -soname="libmylib.so.1" -o libmylib.so.1
```

Refer to the **ld** command in your operating system documentation for more information about the **-shared** and **-soname** options.

Related References

"Compiler Command Line Options" on page 35
"32, 64" on page 45
"mkshrobj" on page 178

prefetch

▶ C ▶ C++

Purpose

Enables generation of prefetching instructions such as dcbt and dcbz in compiled code.

Syntax

```
        ┌─prefetch───┐
▶▶── -q─┴─noprefetch─┴────────────────────────────────────────────▶◀
```

Notes

By default, the compiler may insert prefetch instructions in compiled code. The **-qnoprefetch** option lets you disable this feature.

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

Related References

"Compiler Command Line Options" on page 35

print

▶ C ▶ C++

Purpose

Enables or suppresses listings. Specifying **-qnoprint** overrides all listing-producing options, regardless of where they are specified, to suppress listings.

Syntax

```
              ┌─print───┐
►►── -q───────┴─noprint─┴─────────────────────────────────────────────►◄
```

Notes

The default of **-qprint** enables listings if they are requested by other compiler options. These options are:

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

Example

To compile myprogram.C and suppress all listings, even if some files have **#pragma options source** and similar directives, enter:

```
xlc myprogram.c -qnoprint
```

Related References

"Compiler Command Line Options" on page 35
"attr" on page 59
"list" on page 163
"listopt" on page 164
"source" on page 217
"xref" on page 257

priority

C++

Purpose

Specifies the priority level for the initialization of static objects.

Syntax

```
►►── -q──priority──=──number──────────────────────────────────────────►◄
```

See also “#pragma priority” on page 308 and “#pragma options” on page 299.

Notes

number Is the initialization priority level assigned to the static objects within a file, or the priority level of a shared or non-shared file or library.

You can specify a priority level from 101 (highest priority) to 65535 (lowest priority).

If not specified, the default priority level is 65535.

Example

To compile the file myprogram.C to produce an object file myprogram.o so that objects within that file have an initialization priority of 2000, enter:

```
xlc++ myprogram.C -c -qpriority=2000
```

All objects in the resulting object file will be given an initialization priority of 2000, provided that the source file contains no **#pragma priority(***number***)** directives specifying a different priority level.

Related References

“Compiler Command Line Options” on page 35
“#pragma options” on page 299
“#pragma priority” on page 308

proclocal, procimported, procunknown

▶ C ▶ C++

Purpose

Marks functions as local, imported, or unknown in 64-bit compilations.

Syntax

```
         ┌──────────────────────────────────────────────────────────────┐
         ▼                                                               │
►►──── -q──┬─proclocal───────┬──┬───────────────────────────────┬────────┴──────────────►◄
           ├─noproclocal─────┤  │        ┌─:──────────────┐     │
           ├─procimported────┤  │        ▼                │     │
           ├─noprocimported──┤  └─=──────function_name────┴─────┘
           ├─procunknown─────┤
           └─noprocunknown───┘
```

See also “#pragma options” on page 299.

Default

The default is to assume that all functions whose definition is in the current compilation unit are local **proclocal**, and that all other functions are unknown **procunknown**. If any functions that are marked as local resolve to shared library functions, the linkage editor will detect the error and issue warnings.

Notes

This compiler option applies to 64-bit compilations only.

Available suboptions are:

Local functions
: Local functions are statically bound with the functions that call them. Specifying **-qproclocal** changes the default to assume that all functions are local. **-qproclocal=***names* marks the named functions as local, where *names* is a list of function identifiers separated by colons (**:**). The default is not changed.

  Smaller, faster code is generated for calls to functions marked as local.

Imported functions
: Imported functions are dynamically bound with a shared portion of a library. **-qprocimported** changes the default to assume that all functions are imported. Specifying **-qprocimported=***names* marks the named functions as imported, where *names* is a list of function identifiers separated by colons (**:**). The default is not changed.

  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. If marked functions resolve to statically bound objects, the generated code may be larger and run more slowly than the default code sequence generated for unknown functions.

Unknown functions
: Unknown functions are resolved to either statically or dynamically bound objects during link-editing. Specifying **-qprocunknown** changes the default to assume that all functions are unknown. **-qprocunknown=***names* marks the named functions as unknown, where *names* is a list of function identifiers separated by colons (**:**). The default is not changed.

C++ In C++ programs, function *names* must be specified using their mangled names.

Conflicts among the procedure-marking options are resolved in the following manner:

Options that list function names	The last explicit specification for a particular function name is used.
Options that change the default	This form does not specify a name list. The last option specified is the default for functions not explicitly listed in the name-list form.

Examples

1. 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**, and,
   - function **venus** is specified as **unknown**,

   enter:
   ```
   xlc++ myprogram.c oldprogs.a -qprolocal=fun(int):sun()
     -qprocimported=moon():stars(float) -qprocunknown=venus()
   ```
2. The following example shows typical error messages that result when a function marked as local instead resolves to a shared library function.
   ```
   int main(void)
   {
       printf("Just in function foo1()\n");
       printf("Just in function foo1()\n");
   }
   ```
   Compiling this source code with `xlc -q64 -qproclocal -O -qlist t.c` gives results similar to the following:
   ```
   /opt/cross/bin/powerpc64-linux-ld: t.o(.text+0x10): unresolvable relocation \
     against symbol `.printf@@GLIBC_2.2.5'
       t.o: In function .main':
       t.o(.text+0x10): relocation truncated to fit: R_PPC64_REL24 .printf@@GLIBC_2.2.5
   /opt/cross/bin/powerpc64-linux-ld: t.o(.text+0x18): unresolvable relocation \
     against symbol `.printf@@GLIBC_2.2.5'
       t.o(.text+0x18): relocation truncated to fit: R_PPC64_REL24 .printf@@GLIBC_2.2.5
   ```
   An executable file is produced, but it will not run. The error message indicates that a call to **printf** in object file **t.o** caused the problem. When you have confirmed that the called routine should be imported from a shared object, recompile the source file that caused the warning and explicitly mark **printf** as imported. For example:
   ```
   xlc -c -qprocimported=printf t.c
   ```

Related References

"Compiler Command Line Options" on page 35

proto

C

Purpose

If this option is set, the compiler assumes that all functions are prototyped.

Syntax

```
          ┌─noproto─┐
►►── -q───┴─proto───┴──────────────────────────────────────────►◄
```

Notes

This option asserts that procedure call points agree with their declarations even if the procedure has not been prototyped.

Callers can pass floating-point arguments in floating-point registers only and not in General-Purpose Registers (GPRs). The compiler assumes that the arguments on procedure calls are the same types as the corresponding parameters of the procedure definition.

The compiler will issue warnings for functions that do not have prototypes.

Example

To compile my_c_program.c to assume that all functions are prototyped, enter:

```
xlc my_c_program.c -qproto
```

Related References

"Compiler Command Line Options" on page 35

Q

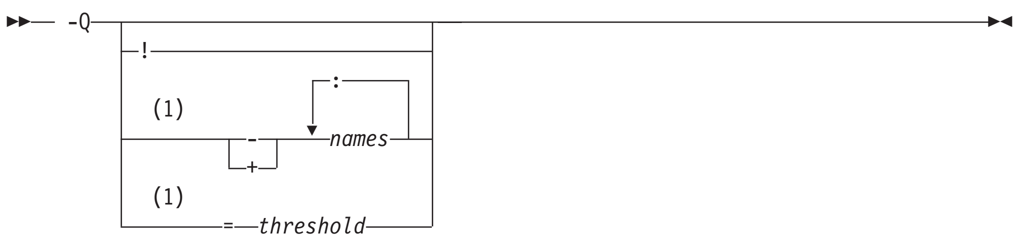
C C++

Purpose

In C++ language applications, this option instructs the compiler to try to inline functions. Inlining is performed if possible but, depending on which optimizations are performed, some functions might not be inlined.

In C language applications, this option specifies which specific functions the compiler should attempt to inline.

Syntax



Notes:

1 C only

C++ In the C++ language, the following **-Q** options apply:

-Q Compiler inlines all functions that it can.
-Q! Compiler does not inline any functions.

C In the C language, the following **-Q** options apply:

-Q
: Attempts to inline all appropriate functions with 20 executable source statements or fewer, subject to the setting of any of the suboptions to the **-Q** option. If **-Q** is specified last, all functions are inlined.

-Q!
: Does not inline any functions. If **-Q!** is specified last, no functions are inlined.

-Q-*names*
: Does not inline functions listed by *names*. Separate each function name in *names* with a colon (**:**). All other appropriate functions are inlined. The option implies **-Q**.

For example:

```
-Q-salary:taxes:expenses:benefits
```

causes all functions except those named **salary**, **taxes**, **expenses**, or **benefits** to be inlined if possible.

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

-Q+*names* Attempts to inline the functions listed by *names* and any other appropriate functions. Each function name in *names* must be separated by a colon (**:**). The option implies **-Q**.

For example,

```
-Q+food:clothes:vacation
```

causes all functions named **food**, **clothes**, or **vacation** to be inlined if possible, along with any other functions eligible for inlining.

A warning message is issued for functions that are not defined in the source file or that are defined but cannot be inlined.

This suboption overrides any setting of the *threshold* value. You can use a threshold value of zero along with **-Q+***names* to inline specific functions. For example:

```
-Q=0
```

followed by:

```
-Q+salary:taxes:benefits
```

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

-Q=*threshold* Sets a size limit on the functions to be inlined. The number of executable statements must be less than or equal to *threshold* for the function to be inlined. *threshold* must be a positive integer. The default value is 20. Specifying a threshold value of **0** causes no functions to be inlined except those functions marked with the **__inline**, **_Inline**, or **_inline** keywords.

The *threshold* value applies to logical C statements. 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 */
  }
}
```

Default

The default is to treat inline specifications as a hint to the compiler. Whether or not inlining occurs may also be dependent on other options that you select:

- If you optimize your programs, (specify the **-O** option) the compiler attempts to inline the functions declared as inline.

Notes

The **-Q** option is functionally equivalent to the **-qinline** option.

If you specify the **-g** option (to generate debug information), inlining may be affected. See the information for the **-g** compiler option.

Because inlining does not always improve run time, you should test the effects of this option on your code.

Do not attempt to inline recursive or mutually recursive functions.

Normally, application performance is optimized if you request optimization (**-O** option), and compiler performance is optimized if you do not request optimization.

The **inline**, **_inline**, **_Inline**, and **__inline** language keywords override all **-Q** options except **-Q!**. The compiler will try to inline functions marked with these keywords regardless of other **-Q** option settings.

To maximize inlining:

- for C programs, specify optimization (**-O**) and also specify the appropriate **-Q** options for the C language.
- for C++ programs, specify optimization (**-O**) but do not specify the **-Q** option.

Examples

To compile the program myprogram.c so that no functions are inlined, enter:

```
xlc myprogram.c -O -Q!
```

To compile the program my_c_program.c so that the compiler attempts to inline functions of fewer than 12 lines, enter:

```
xlc my_c_program.c -O -Q=12
```

Related References

"Compiler Command Line Options" on page 35
"g" on page 107
"inline" on page 125
"O, optimize" on page 179
"Q" on page 199
"The inline, _Inline, _inline, and __inline Function Specifiers" on page 127

R

▶ C ▶ C++

Purpose

At run time, searches the path directory for shared libraries.

Syntax

```
▶▶── -R──directory──────────────────────────────────────────────▶◀
```

Notes

If the **-R***directory* 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**.

Default

The default is to search only the standard directories.

Example

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 References

"Compiler Command Line Options" on page 35
"L" on page 141
"l" on page 142

r

C C++

Purpose

Produces a relocatable object. This permits the output file to be produced even though it contains unresolved symbols.

Syntax

```
►►── -r──────────────────────────────────────────────────────────────►◄
```

Notes

A file produced with this flag is expected to be used as a file parameter in another call to **xlc++**.

Example

To compile myprogram.c and myprog2.c into a single object file **mytest.o**, enter:

```
xlc myprogram.c myprog2.c -r -o mytest.o
```

Related References

"Compiler Command Line Options" on page 35

report

C C++

Purpose

Instructs the compiler to produce transformation reports that show how program loops are parallelized and/or optimized. The transformation reports are included as part of the compiler listing.

Syntax

```
              ┌─noreport─┐
►►── -q───────┴─report───┴─────────────────────────────────────────────►◄
```

Notes

This option has no effect unless **-qhot** or **-qsmp** are also in effect.

Specifying **-qreport** together with **-qhot** instructs the compiler to produce a pseudo-C code listing and summary showing how loops are transformed. You can use this information to tune the performance of loops in your program.

Specifying **-qreport** together with **-qsmp** instructs the compiler to also produce a report showing how the program deals with data and automatic parallelization of loops in your program. You can use this information to determine how loops in your program are or are not parallelized.

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.

Example

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++ -qsmp -O3 -qreport myprogram.C
```

Related References

"Compiler Command Line Options" on page 35
"hot" on page 113
"smp" on page 215

ro

C C++

Purpose

Specifies the storage type for string literals.

Syntax

```
           ┌─ro───┐
►►── -q────┴─noro─┴──────────────────────────────────────────────────►◄
```

See also “#pragma options” on page 299.

Default

The default for all compiler invocations except **cc** and its derivatives is **-qro**. The default for the **cc** compiler invocation is **-qnoro**.

Notes

If **-qro** is specified, the compiler places string literals in read-only storage. If **-qnoro** is specified, string literals are placed in read/write storage.

You can also specify the storage type in your source program using:

```
#pragma strings storage_type
```

where *storage_type* is **read-only** or **writable**.

Placing string literals in read-only memory can improve runtime performance and save storage, but code that attempts to modify a read-only string literal may generate a memory error.

Example

To compile myprogram.c so that the storage type is **writable**, enter:

```
xlc myprogram.c -qnoro
```

Related References

“Compiler Command Line Options” on page 35
“roconst” on page 206
“#pragma options” on page 299
“#pragma strings” on page 316

roconst

C C++

Purpose

Specifies the storage location for constant values.

Syntax

```
           ┌roconst──┐
▶▶── -q────┴noroconst┴──────────────────────────────────────────▶◀
```

See also "#pragma options" on page 299.

Default

The default with **xlc**, **xlC**, and **c89** is **-qroconst**. The default with **cc** is **-qnoroconst**.

Notes

If **-qroconst** is specified, the compiler places constants in read-only storage. If **-qnoroconst** is specified, constant values are placed in read/write storage.

Placing constant values in read-only memory can improve runtime performance, save storage, and provide shared access. Code that attempts to modify a read-only constant value generates a memory error.

Constant value 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 variables do not apply to this option:

- variables qualified with **volatile** and aggregates (such as a **struct** 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**).

Related References

"Compiler Command Line Options" on page 35
"ro" on page 205
"#pragma options" on page 299

rtti

C++

Purpose

Use this option to generate run-time type identification (RTTI) information for exception handling and for use by the typeid and dynamic_cast operators.

Syntax

```
             ┌─nortti─┐
►►── -q──────┴─rtti───┴───────────────────────────────────────────────────►◄
```

where available suboptions are:

rtti	The compiler generates the information needed for the RTTI typeid and dynamic_cast operators.
nortti	The compiler does not generate RTTI information.

Notes

For best run-time performance, suppress RTTI information generation with the default **-qnortti** setting.

The C++ language offers a (RTTI) mechanism for determining the class of an object at run time. It consists of two operators:

- one for determining the run-time type of an object (typeid), and,
- one for doing type conversions that are checked at run time (dynamic_cast).

A type_info class describes the RTTI available and defines the type returned by the typeid operator.

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.

Related References

"Compiler Command Line Options" on page 35
"eh" on page 91

S

▶ C ▶ C++

Purpose

Generates an assembler language file (**.s**) for each source file. The resulting **.s** files can be assembled to produce object **.o** files or an executable file (**a.out**).

Syntax

```
►►— -S————————————————————————————————————————————————————►◄
```

Notes

You can invoke the assembler with the **xlc++** command. For example,

```
xlc++ myprogram.s
```

will invoke the assembler, and if successful, the loader 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
```

Examples

1. To compile myprogram.C to produce an assembler language file **myprogram.s**, enter:

   ```
   xlc++ myprogram.C -S
   ```

2. To assemble this program to produce an object file **myprogram.o**, enter:

   ```
   xlc++ myprogram.s -c
   ```

3. To compile myprogram.C to produce an assembler language file **asmprogram.s**, enter:

   ```
   xlc++ myprogram.C -S -o asmprogram.s
   ```

Related References

"Compiler Command Line Options" on page 35
"E" on page 88
"g" on page 107
"ipa" on page 129
"o" on page 183
"P" on page 184
"tbtable" on page 232

s

▶ C ▶ C++

Purpose

This option strips the symbol table, line number information, and relocation information from the output file. Specifying **-s** saves space, but limits the usefulness of traditional debug programs when you are generating debug information using options such as **-g**.

Syntax

```
▶▶── -s ──────────────────────────────────────────────────────▶◀
```

Notes

Using the strip command has the same effect.

Related References

"Compiler Command Line Options" on page 35
"g" on page 107

saveopt

C C++

Purpose

Saves the command line compiler options into an object file.

Syntax

```
          ┌─nosaveopt─┐
►►── -q───┴─saveopt───┴──────────────────────────────────────────────►◄
```

Notes

This option lets you save command line compiler options into the object file you are compiling. The option has effect only when compiling to an object (**.o**) file.

The string is saved in the following format:

```
►►──@(#)opt──B───┬─f─┬───B──stanza──B──options──────────────────►◄
                 ├─c─┤
                 └─C─┘
```

where:

B Indicates a space.

f Signifies a Fortran language compilation.

c Signifies a C language compilation.

C Signifies a C++ language compilation.

stanza Specifies the driver used for the compilation, for example, c89 or xlc++.

options The list of command line options specified on the command line, with individual options separated by spaces.

Related References

"Compiler Command Line Options" on page 35

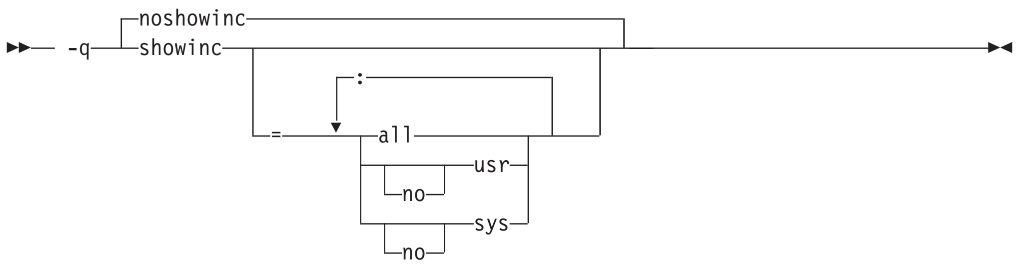
showinc

C C++

Purpose

Used with **-qsource** to selectively show user header files (includes using ″ ″) or system header files (includes using < >) in the program source listing.

Syntax



where options are:

noshowinc	Do not show user or system include files in the program source listing. This is the same as specifying **-qshowinc=nousr:nosys**.
showinc	Show both user and system include files in the program source listing. This is the same as specifying **-qshowinc=usr:sys** or **-qshowinc=all**.
all	Show both user and system include files in the program source listing. This is the same as specifying **-qshowinc** or **-qshowinc=usr:sys**.
usr	Show user include files in the program source listing.
sys	Show system include files in the program source listing.

See also “#pragma options” on page 299.

Notes

This option has effect only when the **-qlist** or **-qsource** compiler options are in effect.

Example

To compile myprogram.C so that all included files appear in the source listing, enter:

```
xlc++ myprogram.C -qsource -qshowinc
```

Related References

“Compiler Command Line Options” on page 35
“source” on page 217
“#pragma options” on page 299

showpdf

C C++

Purpose

Used with **-qpdf1** and a minimum optimization level of **-O** to add additional call and block count profiling information to an executable.

Syntax

```
            ┌noshowpdf┐
►►── -q─────┴showpdf──┴──────────────────────────────────────────►◄
```

Notes

This option has effect only when specified together with the **-qpdf1** compiler option.

When specified with **-qpdf1** and a minimum optimization level of **-O**, the compiler inserts additional profiling information into the compiled application to collect call and block counts for all procedures in the application. Running the compiled application will record the call and block counts to the file **._pdf** .

After you run your application with training data, you can retrieve the contents of the **._pdf** file with the **showpdf** utility. This utility is described in the **-qpdf** pages.

Example

The example assumes the following source for program file `hello.c`:

```
#include <stdio.h>

void HelloWorld()
{
 printf("Hello World");
}

main()
{
 HelloWorld();
}
```

Compile the source with:

```
xlc -qpdf1 -O -qshowpdf hello.c
```

Run the resulting program executable:

```
a.out
```

Run the **showpdf** utility to display the call and block counts for the executable:

```
showpdf
```

Something similar to the following will be returned by the **showpdf** utility:

```
HelloWorld(4):  1 (hello.c)

Call Counters:
  5 | 1  printf(6)

Call coverage = 100% ( 1/1 )

Block Counters:
  3-5 | 1
```

```
  6 |
  6 | 1

Block coverage = 100% ( 2/2 )


-----------------------------------
main(5):  1 (hello.c)

Call Counters:
  10 | 1  HelloWorld(4)

Call coverage = 100% ( 1/1 )

Block Counters:
  8-11 | 1
  11 |

Block coverage = 100% ( 1/1 )

Total Call coverage = 100% ( 2/2 )
Total Block coverage = 100% ( 3/3 )
```

Related References

"Compiler Command Line Options" on page 35
"pdf1, pdf2" on page 187

smallstack

C C++

Purpose

Instructs the compiler to reduce the size of the stack frame.

Syntax

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

Notes

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.

Example

To compile myprogram.c to use a small stack frame, enter:

```
xlc myprogram.c -qipa -qsmallstack
```

Related References

"Compiler Command Line Options" on page 35
"g" on page 107
"ipa" on page 129
"O, optimize" on page 179

smp

C C++

Purpose

Enables parallelization of program code.

Syntax

```
              ┌─nosmp─────────────────────────────────────────────────┐
►►── -q───────┴─smp─────────────────────────────────────────────────┬─┴──────────►◄
                   │     ┌─:──────────────────────────────────────┐ │
                   │     │ ┌─norec_locks─────────────────────────┐ │ │
                   │     │ ├─nonested_par────────────────────────┤ │ │
                   │     │ ├─explicit────────────────────────────┤ │ │
                   │     │ ├─noomp───────────────────────────────┤ │ │
                   │     │ ├─opt─────────────────────────────────┤ │ │
                   │     │ ├─auto────────────────────────────────┤ │ │
                   └─=───▼─┴─noauto──────────────────────────────┴─┴─┘
                           ├─noopt───────────────────────────────┤
                           ├─omp─────────────────────────────────┤
                           ├─noexplicit──────────────────────────┤
                           ├─nested_par──────────────────────────┤
                           ├─rec_locks───────────────────────────┤
                           │                ┌─runtime─────────┐  │
                           └─schedule──=────┴─┬─dynamic──┬─┬───┴──┘
                                              ├─guided───┤ └─=──n─┘
                                              ├─static───┤
                                              └─affinity─┘
```

where:

auto	Enables automatic parallelization and optimization of program code.
noauto	Disables automatic parallelization of program code. Program code explicitly parallelized with SMP or OpenMP pragma statements is optimized.
opt	Enables automatic parallelization and optimization of program code.
noopt	Enables automatic parallelization, but disables optimization of parallelized program code. Use this setting when debugging parallelized program code.
omp	Enables strict compliance to the OpenMP 2.0 standard. Automatic parallelization is disabled. Parallelized program code is optimized. Only OpenMP parallelization pragmas are recognized.
noomp	Enables automatic parallelization and optimization of program code.
explicit	Enables pragmas controlling explicit parallelization of loops.
noexplicit	Disables pragmas controlling explicit parallelization of loops.

nested_par	If specified, nested parallel constructs are not serialized. **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 re-used. This option 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.
nonested_par	Disables parallization of nested parallel constructs.
rec_locks	If specified, recursive locks are used, and nested critical sections will not cause a deadlock.
norec_locks	If specified, recursive locks are not used.
schedule=*sched_type*[**=***n*]	Specifies what kind of scheduling algorithms and chunk size (*n*) are used for loops to which no other scheduling algorithm has been explicitly assigned in the source code. If *sched_type* is not specified, **schedule=runtime** is assumed for the default setting.

Notes

- **-qsmp** must be used only with thread-safe compiler mode invocations such as **xlc_r**. These invocations ensure that the **pthreads**, **xlsmp**, and thread-safe versions of all default run-time libraries are linked to the resulting executable.
- 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** without suboptions is equivalent to specifying **-qsmp=auto:explicit:noomp:norec_locks:nonested_par:schedule=runtime** or **-qsmp=opt:explicit:noomp:norec_locks:nonested_par:schedule=runtime**.
- 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**.
- Specifying **-qsmp** defines the _IBMSMP preprocessing macro.

Related Concepts

"Program Parallelization" on page 11

Related Tasks

"Control Parallel Processing with Pragmas" on page 31

Related References

"Compiler Command Line Options" on page 35
"O, optimize" on page 179
"threaded" on page 237
"Pragmas to Control Parallel Processing" on page 323
"Run-time Options for Parallel Processing" on page 349
"OpenMP Run-time Options for Parallel Processing" on page 352
"Built-in Functions Used for Parallel Processing" on page 354

source

C C++

Purpose

Produces a compiler listing and includes source code.

Syntax

```
              ┌─nosource─┐
►►── -q───────┴─source───┴──────────────────────────────────────────►◄
```

See also “#pragma options” on page 299.

Notes

The **-qnoprint** option overrides this option.

Parts of the source can be selectively printed 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.

Examples

To compile myprogram.C to produce a compiler listing that includes the source for **myprogram.C**, enter:

```
xlc++ myprogram.C -qsource
```

Do not use the **-qsource** compiler option if you want the compiler listing to show only selected parts of your program source. The following code causes the only the source found between the **#pragma options source** and **#pragma options nosource** directives to be included in the compiler listing:

```
#pragma options source
   . . .
/* Source code to be included in the compiler listing
   is bracketed by #pragma options directives.
*/
   . . .
#pragma options nosource
```

Related References

“Compiler Command Line Options” on page 35
“print” on page 194
“#pragma options” on page 299

sourcetype

C C++

Purpose

Instructs the compiler to treat all source files as if they are the source type specified by this option, regardless of actual source filename suffix.

Syntax

```
                        ┌─default───┐
>>── -q──sourcetype──=──┼─c─────────┼──────────────────────────────────><
                        ├─c++───────┤
                        └─assembler─┘
```

where:

default	The compiler assumes that the programming language of a source file will be implied by its filename suffix.
c	The compiler compiles all source files following this option as if they are C language source files.
c++	The compiler compiles all source files following this option as if they are C++ language source files.
assembler	The compiler compiles all source files following this option as if they are Assembler language source files.

Notes

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

The **-qsourcetype** option instructs the compiler to not rely on the filename suffix, and to instead assume a source type as specified by the option.

Ordinarily, the compiler uses the filename 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, a **.C** suffix normally implies C++ source code, and the compiler will treat them as follows:

`hello.c`	The file is compiled as a C file.
`hello.C`	The file is compiled as a C++ file.

This applies whether the file system is case-sensitive or not. However, in a case-insensitive file system, the above two compilations refer to the same physical file. That is, the compiler still recognizes the case difference of the filename argument on the command line and determines the source type accordingly, but will ignore the case when retrieving the file from the file system.

Examples

To treat the source file **hello.C** as being a C language source file, enter:

```
xlc -qsourcetype=c hello.C
```

Related References

"Compiler Command Line Options" on page 35
"+ (plus sign)" on page 43

spill

C C++

Purpose

Specifies the register allocation spill area as being *size* bytes.

Syntax

```
                 ┌─512──┐
►►── -q──spill──=──┴─size─┴──────────────────────────────────────────►◄
```

See also “#pragma options” on page 299.

Notes

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.

Example

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
```

Related References

“Compiler Command Line Options” on page 35
“#pragma options” on page 299

srcmsg

▶ C

Purpose

Adds the corresponding source code lines to the diagnostic messages in the **stderr** file.

Syntax

```
            ┌─nosrcmsg─┐
►►── -q─────┴─srcmsg───┴────────────────────────────────────────────►◄
```

See also "#pragma options" on page 299.

Notes

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. Specifying **-qnosrcmsg** suppresses the generation of both the source line and the finger line, and the error message simply shows the file, line and column where the error occurred.

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.

The default (**-qnosrcmsg**) displays concise messages that can be parsed. Instead of giving the source line and pointers for each error, a single line is displayed, showing the name of the source file with the error, the line and character column position of the error, and the message itself.

Example

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
```

Related References

"Compiler Command Line Options" on page 35
"#pragma options" on page 299

staticinline

▶ C ▶ C++

Purpose

This option controls whether inline functions are treated as static or extern. By default, XL C/C++ treats inline functions as extern.

Syntax

```
          ┌─nostaticinline─┐
▶▶── -q───┴─staticinline───┴─────────────────────────────────────────▶◀
```

Example

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.

```
inline void f() {/*...*/};
```

Using the default, **-qnostaticinline**, gives **f** external linkage.

Related References

"Compiler Command Line Options" on page 35

staticlink

C C++

Purpose

The **-qstaticlink** compiler option controls how shared and non-shared run-time libraries are linked into an application. The XL 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──┘
```

where

nostaticlink	Instructs the compiler not to link statically with libgcc.a
staticlink	Objects generated with this compiler option in effect will link only with static libraries.
libgcc	When this suboption is 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**.

Notes

GNU support for shared and non-shared libraries is controlled by the options shown in the following table.

Option mappings: control of the Linux linker

GNU option	Meaning	XL 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. Ignore when specified with -shared.	-qstaticlink
-shared-libgcc	Use the shared version of libgcc. Ignore when specified with -static.	-qnostaticlink=libgcc
-static-libgcc	Use the static version of libgcc.	-qstaticlink=libgcc

Related References

"Compiler Command Line Options" on page 35

statsym

C C++

Purpose

Adds user-defined, nonexternal names that have a persistent storage class, such as initialized and uninitialized static variables, to the name list (the symbol table of objects).

Syntax

```
           ┌─nostatsym─┐
►►── -q────┴─statsym───┴──────────────────────────────────────────►◄
```

Default

The default is to not add static variables to the symbol table. However, static functions are added to the symbol table.

Example

To compile myprogram.C so that static symbols are added to the symbol table, enter:

```
xlc++ myprogram.C -qstatsym
```

Related References

"Compiler Command Line Options" on page 35

stdinc

C C++

Purpose

Specifies which directories are used for files included by the **#include** <*file_name*> and **#include "***file_name***"** directives. The **-qnostdinc** option excludes the standard include directories from the search path.

Syntax

```
              ┌stdinc───┐
►►── -q───────┴nostdinc─┴──────────────────────────────────────────────►◄
```

See also "#pragma options" on page 299.

Notes

If you specify **-qnostdinc**, the compiler will not search the default search path directories unless you explicitly add them with the **-I***directory* option.

If a full (absolute) path name is specified, this option has no effect on that path name. It will still have an effect on all relative path names.

-qnostdinc is independent of **-qidirfirst**. (**-qidirfirst** searches the directory specified with **-I** *directory* before searching the directory where the current source file resides.

The search order for files is described in *Directory Search Sequence for Include Files Using Relative Path Names*.

The last valid **#pragma options [NO]STDINC** remains in effect until replaced by a subsequent **#pragma options [NO]STDINC**.

Example

To compile myprogram.c so that the directory **/tmp/myfiles** is searched for a file included in myprogram.c with the **#include "myinc.h"** directive, enter:

```
xlc myprogram.c -qnostdinc -I/tmp/myfiles
```

Related References

"Compiler Command Line Options" on page 35
"I" on page 115
"idirfirst" on page 116
"#pragma options" on page 299
"Directory Search Sequence for Include Files Using Relative Path Names" on page 29

strict

C C++

Purpose

Turns off the aggressive optimizations that have the potential to alter the semantics of your program.

Syntax

```
►►── -q──┬─nostrict─┬──────────────────────────────────────────►◄
         └─strict───┘
```

See also "#pragma options" on page 299.

Default

- **-qnostrict** with optimization levels of **-O3** or higher.
- **-qstrict** otherwise.

Notes

-qstrict turns off the following optimizations:

- Performing code motion and scheduling on computations such as loads and floating-point computations that may trigger an exception.
- Relaxing conformance to IEEE rules.
- Reassociating floating-point expressions.

This option is only valid with **-O2** or higher optimization levels.

-qstrict sets **-qfloat=norsqrt**.

-qnostrict sets **-qfloat=rsqrt**.

You can use **-qfloat=rsqrt** to override the **-qstrict** settings.

For example:

- Using **-O3 -qnostrict -qfloat=norsqrt** means that the compiler performs all aggressive optimizations except **-qfloat=rsqrt**.

If there is a conflict between the options set with **-qnostrict** and **-qfloat=***options*, the last option specified is recognized.

Example

To compile myprogram.C so that the aggressive optimizations of **-O3** are turned off, and division by the result of a square root is replaced by multiplying by the reciprocal (**-qfloat=rsqrt**), enter:

```
xlc++ myprogram.C -O3 -qstrict -qfloat=rsqrt
```

Related References

"Compiler Command Line Options" on page 35
"float" on page 99
"O, optimize" on page 179
"#pragma options" on page 299

strict_induction

▶ C ▶ C++

Purpose

Disables loop induction variable optimizations that have the potential to alter the semantics of your program. Such optimizations can 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.

Syntax

```
>>-- -q--+-nostrict_induction-+-------------------------------------><
         '-strict_induction---'
```

Default

- **-qnostrict_induction** with optimization levels 2 or higher.
- **-qstrict_induction** otherwise.

Notes

The specifying option **-O2** implies **-qnostrict_induction**. Specifying both is unnecessary.

Use of option **-qstrict_induction** is generally not recommended because it can cause considerable performance degradation.

Related references

- "O, optimize" on page 179

suspress

C C++

Purpose

Prevents the specified compiler or driver informational or warning messages from being displayed or added to the listings.

Syntax

```
>>-- -q--+-suppress--+-----------------+-+----------------------------------><
         |           |    .-:-------.  | |
         |           |    V         |  | |
         |           '-=----msg_num-+--' |
         |  (1)                          |
         +----------nosuppress-----------+
         |                   .-:-------. |
         |  (2)              V         | |
         '----------nosuppress--=----msg_num-+-'
```

Notes:

1 C only

2 C++ only

Notes

This option suppresses compiler messages only, and has no effect on linker or operating system messages.

To suppress IPA messages, enter **-qsuppress** before **-qipa** on the command line.

Compiler messages that cause compilation to stop, such as (S) and (U) level messages, or other messages depending on the setting of the **-qhalt** compiler option, cannot be suppressed. For example, if the **-qhalt=w** compiler option is set, warning messages will not be suppressed by the **-qsuppress** compiler option.

The **-qnosuppress** compiler option cancels previous settings of **-qsuppress**.

Example

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 References

"Compiler Command Line Options" on page 35
"halt" on page 111
"ipa" on page 129

symtab

C C++

Purpose

Settings for this option determine what information appears in the symbol table.

Syntax

```
►►── -q──symtab──=──┬─unref──┬──────────────────────────────────────────────►◄
                    └─static─┘
```

where:

unref
: Specifies that all **typedef** declarations, **struct**, **union**, and **enum** type definitions are included for processing by the GNU GDB Debugger.

  Use this option with the **-g** option to produce additional debugging information for use with the debugger.

  When you specify the **-g** option, debugging information is included in the object file. To minimize the size of object and executable files, the compiler only includes information for symbols that are referenced. Debugging information is not produced for unreferenced arrays, pointers, or file-scope variables unless **-qsymtab=unref** is specified.

  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 name list.

  The default is to not add static variables to the symbol table.

Examples

To compile myprogram.c so that static symbols are added to the symbol table, enter:

```
xlc myprogram.c -qsymtab=static
```

To include all symbols in myprogram.c in the symbols table for use with a debugger, enter:

```
xlc myprogram.c -g -qsymtab=unref
```

Related References

"Compiler Command Line Options" on page 35
"g" on page 107

syntaxonly

C

Purpose

Causes the compiler to perform syntax checking without generating an object file.

Syntax

```
►►— -q—syntaxonly——————————————————————————————————————————►◄
```

Notes

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 (listings, etc) are still produced if their corresponding options are set.

Examples

To check the syntax of myprogram.c without generating an object file, enter:

```
xlc myprogram.c -qsyntaxonly
```

or

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

Note that in the second example, the **-qsyntaxonly** option overrides the **-o** option so no object file is produced.

Related References

"Compiler Command Line Options" on page 35
"C" on page 63
"c" on page 64
"E" on page 88
"o" on page 183
"P" on page 184

t

C C++

Purpose

Adds the prefix specified by the **-B** option to the designated programs.

Syntax

where programs are:

Program	Description
c	Compiler front end
b	Compiler back end
p	Compiler preprocessor
a	Assembler
I	Interprocedural Analysis tool - compile phase
L	Interprocedural Analysis tool - link phase
l	Linkage editor

Notes

This option must be used together with the **-B** option.

Default

If **-B** is specified but *prefix* is not, the default prefix is **/lib/o**. If **-B***prefix* is not specified at all, the prefix of the standard program names is **/lib/n**.

If **-B** is specified but **-t***programs* is not, the default is to construct path names for all the standard program names.

Example

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 References

"Compiler Command Line Options" on page 35
"B" on page 60

tabsize

C C++

Purpose

Changes the length of tabs as perceived by the compiler.

Syntax

```
                      ┌─8─┐
►►── -q──tabsize──=───┴─n─┴──────────────────────────────────────────►◄
```

where *n* is the number of character spaces representing a tab in your source program.

Notes

This option only affects error messages that specify the column number at which an error occurred. For example, the compiler will consider tabs as having a width of one character if you specify **-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.

Related References

"Compiler Command Line Options" on page 35

tbtable

C C++

Purpose

Generates a traceback table that contains information about each function, including the type of function as well as stack frame and register information. The traceback table is placed in the text segment at the end of its code.

Syntax

```
>>── -q──tbtable──=──┬─none──┬──────────────────────────────────────────><
                     ├─full──┤
                     └─small─┘
```

where suboptions are:

none	No traceback table is generated. The stack frame cannot be unwound so exception handling is disabled.
full	A full traceback table is generated, complete with name and parameter information. This is the default if **-qnoopt** or **-g** are specified.
small	The traceback table generated has no name or parameter information, but otherwise has full traceback capability. This is the default if you have specified optimization and have not specified **-g**.

See also "#pragma options" on page 299.

Notes

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.

Many performance measurement tools require a full traceback table to properly analyze optimized code. The compiler configuration file contains entries to accomodate this requirement. If you do not require full traceback tables for your optimized code, you can save file space by making the following changes to your compiler configuration file:

1. Remove the **-qtbtable=full** option from the **options** lines of the C or C++ compilation stanzas.
2. Remove the **-qtbtable=full** option from the **xlCopt** line of the **DFLT** stanza.

With these changes, the defaults for the **tbtable** option are:

- When compiling with optimization options set, **-qtbtable=small**
- When compiling with no optimization options set, **-qtbtable=full**

Related References

"Compiler Command Line Options" on page 35
"g" on page 107
"O, optimize" on page 179
"#pragma options" on page 299

tempinc

C++

Purpose

Generates separate tempinc files for template functions and class declarations, and places these files in a directory which can be optionally specified.

Syntax

```
              ┌─notempinc──────────────────┐
►►── -q───────┴─tempinc──┬──────────────┬──┴──────────────────────────────────────►◄
                         └─=──directory─┘
```

Notes

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**.

When you specify **-qtempinc**, the compiler assigns a value of 1 to the **__TEMPINC__** macro. This assignment will not occur if **-qnotempinc** has been specified.

Example

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 References

"Compiler Command Line Options" on page 35
"templateregistry" on page 235

See also *Using C++ templates* section of the *XL C/C++ Programming Guide*.

templaterecompile

C++

Purpose

Helps manage dependencies between compilation units that have been compiled using the **-qtemplateregistry** compiler option.

Syntax

```
        ┌─templaterecompile───┐
►►── -q─┴─notemplaterecompile─┴──────────────────────────────►◄
```

Notes

The **-qtemplaterecompile** option helps to manage dependencies between compilation units that have been compiled using the **-qtemplateregistry** option. Given a program in which multiple compilation units reference the same template instantiation, the **-qtemplateregistry** option specifies a single compilation unit to contain the instantiation. No other compilation units will contain this instantiation, and duplication of object code is avoided.

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.

Related References

"Compiler Command Line Options" on page 35
"templateregistry" on page 235
"tempinc" on page 233

See also *Using C++ templates* section of the *XL C/C++ Programming Guide*.

templateregistry

C++

Purpose

Maintains records of all templates as they are encountered in the source and ensures that only one instantiation of each template is made.

Syntax

```
            ┌templateregistry─────────────────────┐
            │                  └=──registry_file┘  │
►►── -q ────┴notemplateregistry────────────────────┴──────────────────►◄
```

Notes

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**.

The **-qtemplateregistry** option 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.

If you do not specify a location, the compiler will save all template registry information to the file **templateregistry** stored in the current working directory. 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 will result in this situation, and may lead to incorrect results.

Example

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 References

"Compiler Command Line Options" on page 35
"templaterecompile" on page 234
"tempinc" on page 233

See also *Using C++ templates* section of the *XL C/C++ Programming Guide*.

tempmax

C++

Purpose

Specifies the maximum number of template include files to be generated by the **-qtempinc** option for each header file.

Syntax

```
                     ┌─1──────┐
▶▶── -q──tempmax──=──┴─number─┴──────────────────────────────────────────▶◀
```

Notes

Specify the maximum number of template files by giving *number* a value between 1 and 99999.

Instantiations are spread among the template include files.

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.

Related References

"Compiler Command Line Options" on page 35
"tempinc" on page 233
"#pragma implementation" on page 280

See also *Using C++ templates* section of the *XL C/C++ Programming Guide*.

threaded

C C++

Purpose

Indicates to the compiler that the program uses multiple threads. Always use this option when compiling or linking multi-threaded applications. This option ensures that all optimizations are thread-safe.

Syntax

```
>>── -q──┬─nothreaded─┬──────────────────────────────────────────────────────><
         └─threaded───┘
```

Default

The default is **-qthreaded** when compiling with **_r** invocation modes, and **-qnothreaded** when compiling with other invocation modes.

Notes

This option applies to both compile and linkage editor 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.

This option does not make code thread-safe, but it will ensure that code already thread-safe will remain so after compile and linking.

Related References

"Compiler Command Line Options" on page 35

tls

C C++

Purpose

Specifies the thread-local storage model to be used by the application.

Syntax

```
>>-- -q--+-tls--+------------------------------+-+------------------><
         |      |           (2)                | |
         |      |     .-----global-dynamic-.   | |
         |      |     |     (1)            |   | |
         |      |     +-----initial-exec---+   | |
         |      '-=---+-local-exec---------+---' |
         |            '-local-dynamic------'     |
         '-notls---------------------------------'
```

Notes:

1 Default if the **-qnopic** compiler option is in effect.

2 Default if the **-qpic** compiler option is in effect.

Notes

This option selects the model used to access thread-local storage.

On systems that support gcc's __thread keyword, the **vac_configure** tool adds **-qtls** to the set of default options in the compiler configuration file.

If **-qtls** is specified without suboptions, the compiler assumes the following settings:

- **-qtls=initial-exec** if **-qnopic** is in effect.
- **-qtls=global-dynamic** if **-qpic** is in effect.

Related References

"Compiler Command Line Options" on page 35
"pic" on page 192

tmplparse

▶ C++

Purpose

This option controls whether parsing and semantic checking are applied to template definitions (class template definitions, function bodies, member function bodies, and static data member initializers) or only to template instantiations. The compiler can check function bodies and variable initializers in template definitions and produce error or warning messages.

Syntax

```
              ┌─no────┐
►►── -q──tmplparse──=──┼─warn──┼──────────────────────────────────────────────►◄
              └─error─┘
```

where suboptions are:

no	Do not parse the template definitions. This reduces the number of errors issued in code written for previous versions of VisualAge C++ and predecessor products. This is the default.
`warn`	Parses template definitions and issues warning messages for semantic errors.
`error`	Treats problems in template definitions as errors, even if the template is not instantiated.

Notes

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, errors found during the parsing or semantic checking of constructs such as the following, always cause error messages:

- return type of a function template
- parameter list of a function template

Related References

"Compiler Command Line Options" on page 35

See also *Using C++ templates* section of the *XL C/C++ Programming Guide*.

tocdata

C C++

Purpose

Marks data as local.

Syntax

```
         ┌─notocdata─┐
►►── -q──┴─tocdata───┴──────────────────────────────────────────────►◄
```

Notes

This option applies only to 64-bit compilations, and is ignored if specified for a 32-bit compilation.

Local variables are statically bound to the functions that use them. **-qtocdata** instructs the compiler to assume that all variables are local.

If an imported variable is assumed to be local, incorrect code may be generated and performance may decrease. Imported variables are dynamically bound to a shared portion of a library. **-qnotocdata** instructs the compiler to assume that all variables are imported.

Conflicts among the data-marking options are resolved in the following manner:

Options that list variable names	The last explicit specification for a particular variable name is used.
Options that change the default	This form does not specify a name list. The last option specified is the default for variables not explicitly listed in the name-list form.

Related References

"Compiler Command Line Options" on page 35

trigraph

C C++

Purpose

Instructs the compiler to recognize trigraph key combinations used to represent characters not found on some keyboards.

Syntax

```
          ┌─trigraph───┐
►►── -q───┴─notrigraph─┴──────────────────────────────────────────────►◄
```

Notes

A trigraph is a combination of three-key character combinations that let you produce a character that is not available on all keyboards.

The trigraph key combinations are:

Key Combination	**Character Produced**
??=	#
??(	[
??)	]
??/	\
??'	^
??<	{
??>	}
??!	|
??-	~

C The default **-qtrigraph** setting can be overridden by explicitly setting the **-q[no]trigraph** option on the command line.

An explicit **-q[no]trigraph** specification on the command line takes precedence over the **-q[no]trigraph** setting normally associated with a given **-qlanglvl** compiler option, regardless of where the **-q[no]trigraph** specification appears on the command line.

C++ The same is true for C++ programs.

Examples

1. To disable trigraph character sequences when compiling your program, enter:

   ```
   xlc++ myprogram.C -qnotrigraph
   ```

Related References

"Compiler Command Line Options" on page 35
"digraph" on page 84
"langlvl" on page 143

tune

▶ C ▶ C++

Purpose

Specifies the architecture system for which the executable program is optimized.

Syntax

```
                 ┌─ppc970─┐
▶▶── -q──tune──=─┴─auto───┴──────────────────────────────────────────▶◀
```

where architecture suboptions are:

`auto`	Produces object code optimized for the platfom on which it is compiled.
`ppc970`	Produces object code optimized for the PowerPC 970 processor.

See also "#pragma options" on page 299.

Default

The default setting of the **-qtune** option depends on the setting of the **-qarch** option. It should also be noted that **-q64** affects the default for **-qarch**, and therefore also affects the default for **-qtune**.

- If **-qtune** is specified without **-qarch**, the compiler uses **-qarch=ppc970**.
- If **-qarch** is specified without **-qtune**, the compiler uses the default tuning option for the specified architecture, and the listing shows: `TUNE=DEFAULT`.

Default **-qtune** settings for specific **-qarch** settings are described in "Acceptable compiler mode and processor architecture combinations" on page 343.

Notes

You can use **-qtune=***suboption* with **-qarch=***suboption*.

- **-qarch=***suboption* specifies the architecture for which the instructions are to be generated.
- **-qtune=***suboption* specifies the target platform for which the code is optimized.

Example

To specify that the executable program testing compiled from `myprogram.C` is to be optimized for a PowerPC 970 hardware platform, enter:

```
xlc++ -o testing myprogram.C -qtune=ppc970
```

Related references

- "Specify Compiler Options for Architecture-Specific, 32- or 64-bit Compilation" on page 25
- "arch" on page 56
- "Acceptable compiler mode and processor architecture combinations" on page 343

U

C C++

Purpose

Undefines the identifier *name* defined by the compiler or by the **-D***name* option.

Syntax

```
▸▸── -U──name──────────────────────────────────────────────────────────▸◂
```

Notes

The **-U***name* 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***name* option.

The identifier name can also be undefined in your source program using the **#undef** preprocessor directive.

The **-U***name* option has a higher precedence than the **-D***name* option.

Example

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 References

"Compiler Command Line Options" on page 35
"D" on page 80

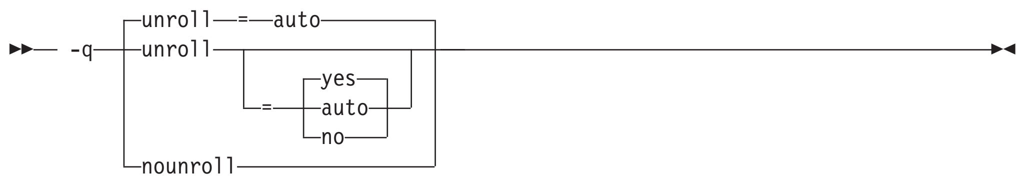
unroll

C C++

Purpose

Unrolls inner loops in the program. This can help improve program performance.

Syntax



where:

-qunroll=auto	Leaves the decision to unroll loops to the compiler. This is the compiler default.
-qunroll or -qunroll=yes	Suggests to the compiler that it unroll loops.
-qnounroll or -qunroll=no	Instructs the compiler to not unroll loops.

See also “#pragma unroll” on page 317 and “#pragma options” on page 299.

Notes

The compiler default for this option, unless explicitly specified otherwise on the command line, is **-qunroll=auto**.

Specifying **-qunroll** without any suboptions is equivalent to specifying **-qunroll=yes**.

When **-qunroll**, **-qunroll=yes**, or **-qunroll=auto** is specified, the bodies of inner loops will be unrolled, or duplicated, by the optimizer. 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.

To see if the **unroll** option improves performance of a particular application, you should first compile the program with usual options, then run it with a representative workload. You should then recompile with command line **-qunroll** option and/or the **unroll** pragmas enabled, then rerun the program under the same conditions to see if performance improves.

You can use the **#pragma unroll** directive to gain more control over unrolling. Setting this pragma overrides the **-qunroll** compiler option setting.

Examples

1. In the following examples, unrolling is disabled:
   ```
   xlc++ -qnounroll file.C

   xlc++ -qunroll=no file.C
   ```
2. In the following examples, unrolling is enabled:

```
xlc++ -qunroll file.C

xlc++ -qunroll=yes file.C

xlc++ -qunroll=auto file.C
```

3. See “#pragma unroll” on page 317 for examples of how program code is unrolled by the compiler.

Related References

“Compiler Command Line Options” on page 35
“#pragma options” on page 299
“#pragma unroll” on page 317

unwind

C C++

Purpose

Informs the compiler that the stack can be unwound while a routine in the compilation is active.

Syntax

```
        ┌─unwind───┐
▶▶── -q─┴─nounwind─┴──────────────────────────────────────────▶◀
```

Notes

Specifying **-qnounwind** can improve optimization of non-volatile register saves and restores.

C++ For C++ programs, specifying **-qnounwind** also implies **-qnoeh**.

Related References

"Compiler Command Line Options" on page 35
"eh" on page 91

upconv

C

Purpose

Preserves the **unsigned** specification when performing integral promotions.

Syntax

```
         ┌─noupconv─┐
►►── -q ─┴─upconv───┴──────────────────────────────────────────────►◄
```

See also “#pragma options” on page 299.

Notes

The **-qupconv** option promotes any **unsigned** type smaller than an **int** to an **unsigned int** instead of to an **int**.

Unsignedness preservation is provided for compatibility with older dialects of C. The ANSI C standard requires value preservation as opposed to unsignedness preservation.

Default

The default is **-qnoupconv**, except when **-qlanglvl=extc89**, in which case the default is **-qupconv**. The compiler does not preserve the **unsigned** specification.

The default compiler action is for integral promotions to convert a **char**, **short int**, **int** bit field or their **signed** or **unsigned** types, or an **enumeration** type to an **int**. Otherwise, the type is converted to an **unsigned int**.

Example

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 References

“Compiler Command Line Options” on page 35
“langlvl” on page 143

utf

C C++

Purpose

Enables recognition of UTF literal syntax.

Syntax

```
        ┌─noutf─┐
►►── -q─┴─utf───┴──────────────────────────────────────────────►◄
```

Notes

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.

Related References

"Compiler Command Line Options" on page 35

V

C C++

Purpose

Instructs the compiler to report information on the progress of the compilation, names the programs being invoked within the compiler and the options being specified to each program. Information is displayed in a space-separated list.

Syntax

```
►►── -V──────────────────────────────────────────────────────────────►◄
```

Notes

The **-V** option is overridden by the **-#** option.

Example

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 References

"Compiler Command Line Options" on page 35
"# (pound sign)" on page 44
"v" on page 250

v

C C++

Purpose

Instructs the compiler to report information on the progress of the compilation, names the programs being invoked within the compiler and the options being specified to each program. Information is displayed in a comma-separated list.

Syntax

```
►►— -v————————————————————————————————————————————————————————►◄
```

Notes

The **-v** option is overridden by the **-#** option.

Example

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 References

"Compiler Command Line Options" on page 35
"# (pound sign)" on page 44
"V" on page 249

vftable

C++

Purpose

Controls the generation of virtual function tables.

Syntax

```
          ┌─vftable───┐
►►── -q───┴─novftable─┴──────────────────────────────────────────►◄
```

Default

The default is to define the virtual function table for a class if the current compilation unit contains the body of the first non-inline virtual member function declared in the class member list.

Notes

Specifying **-qvftable** generates virtual function tables for all classes with virtual functions that are defined in the current compilation unit.

If you specify **-qnovftable**, no virtual function tables are generated in the current compilation unit.

Example

To compile the file myprogram.C so that no virtual function tables are generated, enter:

```
xlc++ myprogram.C -qnovftable
```

Related References

"Compiler Command Line Options" on page 35

vrsave

C C++

Purpose

Enables code in function prologs and epilogs to maintain the VRSAVE register.

Syntax

```
          ┌─vrsave───┐
►►── -q───┴─novrsave─┴───────────────────────────────────────────►◄
```

where:

vrsave	Prologs and epilogs of functions in the compilation unit include code needed to maintain the VRSAVE register.
novrsave	Prologs and epilogs of functions in the compilation unit do not include code needed to maintain the VRSAVE register.

Notes

Use **#pragma altivec_vrsave** to override the current setting of this compiler option for individual functions within your program source.

Related References

"Compiler Command Line Options" on page 35
"#pragma altivec_vrsave" on page 263

W

C C++

Purpose

Passes the listed option to a designated compiler program.

Syntax

```
                          ┌──────────────┐
                          ▼              │
►►── -W──┬─a─┬──────,──option──┴────────────────────────────────────────►◄
         ├─b─┤
         ├─c─┤
         ├─I─┤
         ├─L─┤
         ├─l─┤
         └─p─┘
```

where programs are:

program	Description
a	Assembler
b	Compiler back end
c	Compiler front end
I	Interprocedural Analysis tool - compile phase
L	Interprocedural Analysis tool - link phase
l	linkage editor
p	compiler preprocessor

Notes

When used in the configuration file, the **-W** option accepts the escape sequence backslash comma (**\,**) to represent a comma in the parameter string.

Examples

1. To compile myprogram.c so that the *option* **-pg** is passed to the linkage editor (**l**) and the assembler (**a**), enter:

   ```
   xlc myprogram.c -Wl,-pg -Wa,-pg
   ```

2. In a configuration file, use the **\,** sequence to represent the comma (,).

   ```
   -Wl\,-pg,-Wa\,-pg
   ```

Related References

"Compiler Command Line Options" on page 35

w

C C++

Purpose

Requests that warnings and lower-level messages be suppressed. Specifying this option is equivalent to specifying **-qflag=e:e**.

Syntax

```
►►── -w──────────────────────────────────────────────────────────────►◄
```

Notes

Informational and warning messages that supply additional information to a severe error are not disabled by this option. For example, a severe error caused by problems with overload resolution will also produce information messages. These informational messages are not disabled with **-w** option:

```
void func(int a){}
void func(int a, int b){}
int main(void)
{
func(1,2,3);
return 0;
}
```

```
"x.cpp", line 6.4: 1540-0218 (S) The call does not match any parameter list for "func".
"x.cpp", line 1.6: 1540-1283 (I) "func(int)" is not a viable candidate.
"x.cpp", line 6.4: 1540-0215 (I) The wrong number of arguments have been specified for "func(int)".
"x.cpp", line 2.6: 1540-1283 (I) "func(int, int)" is not a viable candidate.
"x.cpp", line 6.4: 1540-0215 (I) The wrong number of arguments have been specified for "func(int, int)".
```

Example

To compile myprogram.c so that no warning messages are displayed, enter:

```
xlc++ myprogram.C -w
```

Related References

"Compiler Command Line Options" on page 35
"flag" on page 98

warn64

C C++

Purpose

Enables checking for possible data conversion problems between 32-bit and 64-bit compiler modes.

Syntax

```
          ┌─nowarn64─┐
►►── -q───┴─warn64───┴──────────────────────────────────────────►◄
```

Notes

All generated messages have level Informational.

The **-qwarn64** option functions in either 32- or 64-bit compiler modes. In 32-bit mode, it functions as a preview aid to discover possible 32- to 64-bit migration problems.

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
- conflicts with pragma options **arch** in source files and on the command line

Related References

"Compiler Command Line Options" on page 35
"32, 64" on page 45

xcall

C C++

Purpose

Generates code to treat static routines within a compilation unit as if they were external routines.

Syntax

```
             ┌─noxcall─┐
►►── -q──────┴─xcall───┴────────────────────────────────────────────►◄
```

Notes

-qxcall generates slower code than **-qnoxcall**.

Example

To compile myprogram.c so that all static routines are compiled as external routines, enter:

```
xlc myprogram.c -qxcall
```

Related References

“Compiler Command Line Options” on page 35

xref

▶ C ▶ C++

Purpose

Produces a compiler listing that includes a cross-reference listing of all identifiers.

Syntax

where:

noxref	Do not report identifiers in the program.
xref	Reports only those identifiers that are used.
xref=full	Reports all identifiers in the program.

See also “#pragma options” on page 299.

Notes

The **-qnoprint** option overrides this option.

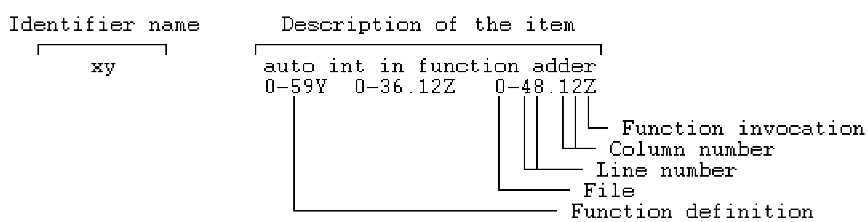
Any function defined with the **#pragma mc_func** *function_name* directive is listed as being defined on the line of the **#pragma** directive.

Example

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 -qattr
```

A typical cross-reference listing has the form:



Related References

“Compiler Command Line Options” on page 35
“attr” on page 59
“print” on page 194
“#pragma mc_func” on page 295
“#pragma options” on page 299

y

C C++

Purpose

Specifies the compile-time rounding mode of constant floating-point expressions.

Syntax

```
            ┌─n─┐
►►── -y─────┼─m─┼──────────────────────────────────────────────────►◄
            ├─p─┤
            └─z─┘
```

where suboptions are:

n	Round to the nearest representable number. This is the default.
m	Round toward minus infinity.
p	Round toward plus infinity.
z	Round toward zero.

Example

To compile myprogram.c so that constant floating-point expressions are rounded toward zero at compile time, enter:

```
xlc myprogram.c -yz
```

Related References

"Compiler Command Line Options" on page 35

General Purpose Pragmas

The pragmas listed below are available for general programming use. Unless noted otherwise, pragmas can be used in both C and C++ programs.

Language Application	#pragma	Description
C C++	#pragma align	Aligns data items within structures.
C	#pragma alloca	Provides an inline version of the function **alloca(size_t size)**.
C C++	#pragma altivec_vrsave	Adds code to function prologs and epilogs to maintain the VRSAVE register.
C C++	#pragma block_loop	Instructs the compiler to create a blocking loop for a specific loop in a loop nest.
C C++	#pragma chars	Sets the sign type of character data.
C C++	#pragma comment	Places a comment into the object file.
C C++	#pragma complexgcc	Instructs the compiler how to pass parameters when calling complex math functions.
C++	#pragma define	Forces the definition of a template class without actually defining an object of the class.
C C++	#pragma disjoint	Lists the identifiers that are not aliased to each other within the scope of their use.
C++	#pragma do_not_instantiate	Suppresses instantiation of a specified template declaration.
C C++	#pragma enum	Specifies the size of enum variables that follow.
C C++	#pragma execution_frequency	Lets you mark program source code that you expect will be either very frequently or very infrequently executed.
C++	#pragma hashome	Informs the compiler that the specified class has a home module that will be specified by the **IsHome** pragma.
C C++	#pragma ibm snapshot	Allows the user to specify a location at which a breakpoint can be set and to define a list of variables that can be examined when program execution reaches that location.
C++	#pragma implementation	Tells the compiler the name of the file containing the function-template definitions that correspond to the template declarations in the include file which contains the pragma.
C C++	#pragma info	Controls the diagnostic messages generated by the **info(...)** compiler options.
C++	#pragma instantiate	Causes immediate instantiation of a specified template declaration.
C++	#pragma ishome	Informs the compiler that the specified class's home module is the current compilation unit.
C C++	#pragma isolated_call	Marks a function that does not have or rely on side effects, other than those implied by its parameters.
C	#pragma langlvl	Selects the C or C++ language level for compilation.

Language Application	#pragma	Description
C C++	#pragma leaves	Takes a function name and specifies that the function never returns to the instruction after the function call.
C C++	#pragma loop_id	Marks a block with a scope-unique identifier.
C C++	#pragma map	Tells the compiler that all references to an identifier are to be converted to a new name.
C C++	#pragma mc_func	Lets you define a function containing a short sequence of machine instructions.
C C++	#pragma nosimd	Instructs the compiler to *not* generate VMX (Vector Multimedia Extension) instructions in the **for** loop immediately following this directive.
C C++	#pragma novector	Instructs the compiler to *not* auto-vectorize the next loop.
C C++	#pragma options	Specifies options to the compiler in your source program.
C C++	#pragma option_override	Specifies alternate optimization options for specific functions.
C C++	#pragma pack	Modifies the current alignment rule for members of structures that follow this pragma.
C++	#pragma priority	Specifies the order in which static objects are to be initialized.
C C++	#pragma reachable	Declares that the point after the call to a routine marked reachable can be the target of a branch from some unknown location.
C C++	#pragma reg_killed_by	Specifies those registers which value will be corrupted by the specified function. It must be used together with **#pragma mc_func**.
C++	#pragma report	Controls the generation of specific messages.
C C++	#pragma stream_unroll	Breaks a stream contained in a loop into multiple streams.
C C++	#pragma strings	Sets storage type for strings.
C C++	#pragma unroll	Unrolls innermost and outermost loops in the program. This can help improve program performance.
C C++	#pragma unrollandfuse	Instructs the compiler to attempt an unroll and fuse operation on nested **for** loops. This can help improve program performance.
C C++	#pragma weak	Prevents the link editor from issuing error messages if it does not find a definition for a symbol, or if it encounters a symbol multiply-defined during linking.

Related Tasks

"Specify Compiler Options in Your Program Source Files" on page 23

#pragma align

C C++

Description

The **#pragma align** directive specifies how the compiler should align data items within structures.

Syntax

```
►►─#─pragma─align─(──┬─linuxppc───┬─)─────────────────────────────────────►◄
                     ├─bit_packed─┤
                     └─reset──────┘
```

See also “#pragma options” on page 299.

Notes

The **#pragma align=***suboption* directive overrides the **-qalign** compiler option setting for a specified section of program source code.

The compiler stacks alignment directives, so you can go back to using a previous alignment directive without knowing what it is by specifying the **#pragma align(reset)** directive.

For example, you can use this option if you have a class declaration within an include file and you do not want the alignment rule specified for the class to apply to the file in which the class is included. You can code **#pragma align(reset)** in a source file to change the alignment option to what it was before the last alignment option was specified. If no previous alignment rule appears in the file, the alignment rule specified in the invocation command is used.

Specifying **#pragma align** has the same effect as specifying **#pragma options align** in your source file. For more information and examples of **#pragma align** and **#pragma options align** usage, see “align” on page 50.

Related references

- See also "Aligning data in aggregates" in *XL C/C++ Programming Guide*.
- See also "The aligned variable attribute" and "The packed variable attribute" in *XL C/C++ Language Reference*.

#pragma alloca

C

Description

The **#pragma alloca** directive specifies that the compiler should provide an inline version of the function **alloca(size_t***size***)**. The function **alloca(size_t***size***)** can be used to allocate space for an object. The amount of space allocated is determined by the value of *size*, which is measured in bytes. The allocated space is put on the stack.

Syntax

```
▶▶──#──pragma──alloca──────────────────────────────────────────────────▶◀
```

Notes

You must specify the **#pragma alloca** directive or the **-qalloca** compiler option to have the compiler provide an inline version of alloca.

Once specified, **#pragma alloca** applies to the rest of the file and cannot be turned off. If a source file contains any functions that you want compiled without **#pragma alloca**, place these functions in a different file.

Related References

"General Purpose Pragmas" on page 259
"alloca" on page 54

#pragma altivec_vrsave

▶ C ▶ C++

Descripton

When the **#pragma altivec_vrsave** directive is enabled, function prologs and epilogs include code to maintain the VRSAVE register.

Syntax

```
                                  ┌on───┐
▶▶──#──pragma──altivec_vrsave─────┼off──┼─────────────────────────────────────────▶◀
                                  └allon┘
```

where pragma settings do the following:

on — Function prologs and epilogs include code to maintain the VRSAVE register.

off — Function prologs and epilogs do not include code to maintain the VRSAVE register.

allon — The function containing the **altivec_vrsave** pragma sets all bits of the VRSAVE register to 1, indicating that all vectors are used and should be saved if a context switch occurs.

Notes

Each bit in the VRSAVE register corresponds to a vector register, and if set to 1 indicates that the corresponding vector register contains data to be saved when a context switch occurs.

This pragma can be used only within a function, and its effects apply only to the function in which it appears. Specifying this pragma with different settings within the same function will create an error condition.

Related References

"General Purpose Pragmas" on page 259
"altivec" on page 55
"vrsave" on page 252

#pragma block_loop

C C++

Description

Marks a block with a scope-unique identifier.

Syntax

```
                                      ┌─,─────────┐
                                      ▼           │
►►─#─pragma─block_loop─(─n─,─────name_list─┴─)──────────────────►◄
```

where:

n Is an integer expression the size of the iteration group.

name_list Is a unique identifier you can create using the **#pragma loopid** directive. If you do not specify *name*, blocking occurs on the first **for** loop following the **#pragma block_loop** directive.

name is an identifier that is unique within the scoping unit.

Notes

For loop blocking to occur, a **#pragma blockloop** directive must precede a **for** loop.

You must not specify **#pragma blockloop** more than once, or combine the directive with the **nounroll**, **unroll**, **nounrollandfuse**, **unrollandfuse**, or **stream_unroll** #pragma directives for the same **for** loop.

You must not specify **#pragma blockloop** more than once for a given loop.

Related References

"General Purpose Pragmas" on page 259
"unroll" on page 244
"#pragma unroll" on page 317
"#pragma unrollandfuse" on page 319
"#pragma stream_unroll" on page 314

#pragma chars

▶ C ▶ C++

Descripton

The **#pragma chars** directive sets the sign type of char objects to be either **signed** or **unsigned**.

Syntax

```
                              ┌─unsigned─┐
▶▶──#──pragma──chars──(──┴─signed───┴──)─────────────────────────────────▶◀
```

Notes

In order to have effect, this pragma must appear before any source statements.

Once specified, the pragma applies to the entire file and cannot be turned off. If a source file contains any functions that you want to be compiled without **#pragma chars**, place these functions in a different file. If the pragma is specified more than once in the source file, the first one will take precedence.

Note: The default character type behaves like an unsigned char.

Related References

"General Purpose Pragmas" on page 259
"chars" on page 68

#pragma comment

C C++

Description

The **#pragma comment** directive places a comment string into the target or object file.

Syntax

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

where suboptions do the following:

compiler	The name and version of the compiler is appended to the end of the generated object module.
date	The date and time of compilation is appended to the end of the generated object module.
timestamp	The date and time of the last modification of the source is appended to the end of the generated object module.
copyright	The text specified by the *token_sequence* is placed by the compiler into the generated object module and is loaded into memory when the program is run.
user	The text specified by the *token_sequence* is placed by the compiler into the generated object but is *not* loaded into memory when the program is run.

Example

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;
}
```

You can use the operating system **strings** command to look for these and other strings in an object or binary file. 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 10:28:09 2004
XL C/C++ for Linux Compiler Version 7.0
Mon Mar 1 10:28:13 2004
My copyright
```

Related References

"General Purpose Pragmas" on page 259

#pragma complexgcc

C C++

Description

The **#pragma complexgcc** directive instructs the compiler how to pass parameters when calling complex math functions.

Syntax

```
►►—#—pragma—complexgcc—(—┬—on——┬—)——————————————————————————————►◄
                          ├—off—┤
                          └—pop—┘
```

where suboptions do the following:

on
: Pushes **-qfloat=complexgcc** onto the stack. This instructs the compiler to use GCC conventions when passing and returning complex parameters.

off
: Pushes **-qfloat=nocomplexgcc** onto the stack. This instructs the compiler to use AIX conventions when passing and returning complex parameters.

pop
: Removes the current setting from the stack, and restores the previous setting. If the stack is empty, the compiler will assume the **-qfloat=[no]complexgcc** setting specified on the command line, or if not specified, the compiler default for **-qfloat=[no]complexgcc**.

Notes

The current setting of this pragma affects only functions declared or defined while the setting is in effect. It does not affect other functions.

Calling functions through pointers to functions will always use the convention set by the **-qfloat=[no]complexgcc** compiler option. If this option is not explicitly set on the command line when invoking the compiler, the compiler default for this option is used. An error will result if you mix and match functions that pass complex values by value or return complex values.

For example, assume the following code is compiled with **-qfloat=nocomplexgcc**:

```
#pragma complexgcc(on)
void p (_Complex double x) {}

#pragma complexgcc(pop)
typedef void (*fcnptr) (_Complex double);

int main() {
    fcnptr ptr = p; /* error: function pointer is -qfloat=nocomplexgcc;
                               function is -qfloat=complexgcc */
}
```

Related References

"General Purpose Pragmas" on page 259
"complexgccincl" on page 73
"float" on page 99

#pragma define

C++

Description

The **#pragma define** directive forces the definition of a template class without actually defining an object of the class. This pragma is only provided for backward compatibility purposes.

Syntax

```
►►—#—pragma—define—(—template_classname—)——————————————————►◄
```

where the *template_classname* is the name of the template to be defined.

Notes

A user can explicitly instantiate a class, function or member template specialization by using a construct of the form:

```
template declaration
```

For example:

```
#pragma define(Array<char>)
```

is equivalent to:

```
template class Array<char>;
```

This pragma must be defined in namespace scope (i.e. it cannot be enclosed inside a function/class body). It is used when organizing your program for the efficient or automatic generation of template functions.

Related References

"General Purpose Pragmas" on page 259
"#pragma do_not_instantiate" on page 270
"#pragma instantiate" on page 284

#pragma disjoint

C C++

Description

The **#pragma disjoint** directive lists the identifiers that are not aliased to each other within the scope of their use.

Syntax

```
                                         ┌──────────────────────────────┐
                                         ▼                              │
►►─#─pragma─disjoint─(─┬───┬─identifier───,─┬───┬─identifier─┴─)──────────►◄
                       └─*─┘                 └─*─┘
```

Notes

The directive informs the compiler that none of the identifiers listed shares the same physical storage, which provides more opportunity for optimizations. If any identifiers actually share physical storage, the pragma may cause the program to give incorrect results.

An identifier in the directive must be visible at the point in the program where the pragma appears. The identifiers in the disjoint name list cannot refer to any of the following:

- a member of a structure, or union
- a structure, union, or enumeration tag
- an enumeration constant
- a typedef name
- a label

This pragma can be disabled with the **-qignprag** compiler option.

Example

```
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
void one_function()
{
     b = 6;
     *ptr_a = 7; // Assignment does not alter the value of b
     another_function(b); // Argument "b" has the value 6
}
```

Because external pointer *ptr_a* does not share storage with and never points to the external variable *b*, the assignment of 7 to the object that *ptr_a* points to 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 of *another_function* has the value 6 and will not reload the variable from memory.

Related References

"General Purpose Pragmas" on page 259
"ignprag" on page 118
"alias" on page 48

#pragma do_not_instantiate

C++

Description

The **#pragma do_not_instantiate** directive instructs the compiler to *not* instantiate the specified template declaration.

Syntax

```
▶▶──#──pragma──do_not_instantiate──template──────────────────────────▶◀
```

where *template* is a class template-id. For example:

```
#pragma do_not_instantiate Stack < int >
```

Notes

Use this pragma to suppress the implicit instantiation of a template for which a definition is supplied.

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 link-edit step.

Related References

"General Purpose Pragmas" on page 259
"#pragma define" on page 268
"#pragma instantiate" on page 284
"tempinc" on page 233
"templateregistry" on page 235

#pragma enum

C C++

Description

The **#pragma enum** directive specifies the size of enum variables that follow. The size at the left brace of a declaration is the one that affects that declaration, regardless of whether further enum directives occur within the declaration. This pragma pushes a value on a stack each time it is used, with a reset option available to return to the previously pushed value.

Syntax

```
>>--#--pragma--enum--+-(--suboption--)-+-----------------------------------><
                     '-=--suboption----'
```

where *suboption* is any of the following:

1	The enumeration type is one byte in length, of type **char** if the range of enumeration values falls within the limits of **signed char**, and **unsigned char** otherwise.
2	The enumeration type is two bytes in length, of type **short** if the range of enumeration values falls within the limits of **signed short**, and **unsigned short** otherwise.
4	The enumeration type is four bytes in length, of type **int** if the range of enumeration values falls within the limits of **signed int**, and **unsigned int** otherwise.
8	The enumeration type is eight bytes in length. In 32-bit compilation mode, the enumeration is of type **long long** if the range of enumeration values falls within the limits of **signed long long**, and **unsigned long long** otherwise. In 64-bit compilation mode, the enumeration is of type **long** if the range of enumeration values falls within the limits of **signed long**, and **unsigned long** otherwise.
int	Same as **#pragma enum=4**.
intlong	Specifies that enumeration will occupy 8 bytes of storage if the range of values in the enumeration exceeds the limit for **int**. See the description for **#pragma enum=8**. 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	The enumeration type is the smallest integral type that can contain all variables. If an 8-byte enum results, the actual enumeration type used is dependent on compilation mode. See the description for **#pragma enum=8**.
pop	This suboption resets the enum size setting to its previous **#pragma enum** setting. If there is no previous setting, the command line setting for **-qenum** is used.
reset	Same as **pop**. This option is provided for backwards compatibility.

Notes

Popping on an empty stack generates a warning message and the enum value remains unchanged.

The **#pragma enum** directive overrides the **-qenum** compiler option.

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 is the only way to prevent one file from potentially changing the **enum** setting of another file that **#include**s it.

The **#pragma options enum=** directive can be used instead of **#pragma enum**. The two pragmas are interchangeable.

A **-qenum=reset** option corresponding to the **#pragma enum=reset** directive does not exist. Attempting to use **-qenum=reset** generates a warning message and the option is ignored.

Examples

1. Usage of the **pop** and **reset** suboptions are shown in the following code segment.

```
#pragma enum(1)
#pragma enum(2)
#pragma enum(4)
#pragma enum(pop)    /* will reset enum size to 2       */
#pragma enum(reset)  /* will reset enum size to 1       */
#pragma enum(pop)    /* will reset enum size to the -qenum setting,
                        assuming -qenum was specified on the command
                        line.  If -qenum was not specified on the
                        command line, the compiler default is used. */
```

2. One typical use for the **reset** suboption is to reset the enumeration size set at the end of an include file that specifies an enumeration storage different from the default in the main file. For example, the following include file, small_enum.h, declares various minimum-sized enumerations, then resets the specification at the end of the include file to the last value on the option stack:

```
#ifndef small_enum_h
#define small_enum_h 1
/*
 * File small_enum.h
 * This enum must fit within an unsigned char type
 */

#pragma options enum=small
enum e_tag {a, b=255};
enum e_tag u_char_e_var; /* occupies 1 byte of storage */


/* Reset the enumeration size to whatever it was before */
#pragma options enum=reset
#endif
```

The following source file, int_file.c, includes small_enum.h:

```
/*
 * File int_file.c
 * Defines 4 byte enums
 */
#pragma options enum=int
enum testing {ONE, TWO, THREE};
enum testing test_enum;

/* various minimum-sized enums are declared */
#include "small_enum.h"

/* return to int-sized enums. small_enum.h has reset the
 * enum size
 */
enum sushi {CALIF_ROLL, SALMON_ROLL, TUNA, SQUID, UNI};
enum sushi first_order = UNI;
```

The enumerations **test_enum** and **first_order** both occupy 4 bytes of storage and are of type **int**. The variable **u_char_e_var** defined in small_enum.h occupies 1 byte of storage and is represented by an **unsigned char** data type.

3. If the following C fragment is compiled with the **enum=small** option:
```
enum e_tag {a, b, c} e_var;
```
the range of enum 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**.

4. If the following C code fragment is compiled with the **enum=small** option:
```
enum e_tag {a=-129, b, c} e_var;
```
the range of enum 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**.

5. If you compile a file myprogram.C using the command:
```
xlc++ myprogram.C -qenum=small
```
assuming file myprogram.C does not contain **#pragma options=int** statements, all **enum** variables within your source file will occupy the minimum amount of storage.

6. If you compile a file yourfile.C that contains the following lines:
```
enum testing {ONE, TWO, THREE};
enum testing test_enum;

#pragma options enum=small
enum sushi {CALIF_ROLL, SALMON_ROLL, TUNA, SQUID, UNI};
enum sushi first_order = UNI;

#pragma options enum=int
enum music {ROCK, JAZZ, NEW_WAVE, CLASSICAL};
enum music listening_type;
```
using the command:
```
xlc++ yourfile.C
```
only the enum variable **first_order** will be minimum-sized (that is, enum variable **first_order** will only occupy 1 byte of storage). The other two enum variables **test_enum** and **listening_type** will be of type **int** and occupy 4 bytes of storage.

The following examples show invalid enumerations or usage of **#pragma enum**:

- You cannot change the storage allocation of an enum using a **#pragma enum** within the declaration of an enum. The following code segment generates a warning and the second occurrence of the **enum** option 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
```

- The **enum** constant range does not fit within the range of an **unsigned int**.

```
#pragma options enum=small
enum e_tag { a=0,
             b=4294967296 /* error: larger than maximum int */
           } e_var;
#pragma options enum=reset
```

Related References

"General Purpose Pragmas" on page 259
"enum" on page 93
"#pragma options" on page 299

#pragma execution_frequency

C C++

Description

The **#pragma execution_frequency** directive lets you mark program source code that you expect will be either very frequently or very infrequently executed.

Syntax

```
>>—#—pragma—execution_frequency—(—┬very_low──┬—)——————————————————————><
                                  └very_high─┘
```

Notes

Use this pragma to mark program source code that you expect will be executed very frequently or very infrequently. The pragma must be placed within block scope, and acts on the closest point of branching.

The pragma is used as a hint to the optimizer. If optimization is not selected, this pragma has no effect.

Examples

1. This 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();
   }
   ```

   The code block ″Block B″ would be 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();
   }
   ```

2. This 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.   */
   ```

3. This pragma cannot be used at file scope. It can be placed anywhere within a block scope and it 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)
```

The first and fourth pragmas are invalid, while the second and third are valid. However, only the third pragma has effect, and it affects whether program execution branches to Block A or Block B during the decision of ″**if (boo)**″. The second pragma is ignored by the compiler.

Related References

“General Purpose Pragmas” on page 259

#pragma hashome

C++

Description

The **#pragma hashome** directive 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** was specified.

Syntax

```
>>--#--pragma--hashome--(--className--+------------+--)------------------------><
                                      '-AllInlines-'
```

where:

className specifies the name of a class that requires the above mentioned external referencing. *className* must be a class and it must be defined.

AllInlines specifies that all inline functions from within *className* should be referenced as being external. This argument is case insensitive.

Notes

A warning will be produced if there is a **#pragma ishome** without a matching **#pragma hashome**.

Example

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 References

"General Purpose Pragmas" on page 259
"#pragma ishome" on page 285

#pragma ibm snapshot

C C++

Description

The **#pragma ibm snapshot** allows the user to specify a location at which a breakpoint can be set and to define a list of variables that can be examined when program execution reaches that location.

Syntax

```
                                   ┌─,────────────┐
                                   ▼              │
►►──#──pragma──ibm snapshot──(───variable_name──┴──)──────────────────────────────►◄
```

where *variable_name* is a collection of variables. Class, structure, or union members cannot be specified.

Notes

This pragma is provided to facilitate debugging optimized code produced by the XL C/C++ compiler. During a debugging session, a breakpoint can be placed on this line to view the values of the named variables. When the program has been compiled with optimization and including the option **-g**, the named variables are guaranteed to be visible to the debugger.

Variables specified in **#pragma ibm snapshot** 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.

Example

```
#pragma ibm snapshot(a, b, c)
```

If a breakpoint is set through the debugger at this point in a program, the values of variables `a`, `b`, and `c` should be visible.

Related references

- "g" on page 107
- "O, optimize" on page 179

#pragma implementation

▶ C++

Description

The **#pragma implementation** directive tells the compiler the name of the template instantiation file containing the function-template definitions. These definitions correspond to the template declarations in the #include file containing the pragma.

Syntax

```
▶▶—#—pragma—implementation—(—string_literal—)————————————————————▶◀
```

Notes

This pragma can appear anywhere that a declaration is allowed. It is used when organizing your program for the efficient or automatic generation of template functions.

Related References

"General Purpose Pragmas" on page 259
"tempmax" on page 236

#pragma info

C C++

Description

The **#pragma info** directive instructs the compiler to produce or suppress specific groups of compiler messages.

Syntax

```
►►─#─pragma─info─(─┬─all─────────┬─)─────────────────────────────────►◄
                   ├─none────────┤
                   ├─restore─────┤
                   │  ┌─,─────┐  │
                   │  ▼       │  │
                   └───group──┴──┘
```

where:

`all`	Turns on all diagnostic checking.
`none`	Turns off all diagnostic suboptions for specific portions of your program.
`restore`	Restores the option that was in effect before the previous **#pragma info** directive.

group Generates or suppresses all messages associated with the specified diagnostic *group*. More than one *group* name in the following list can be specified.

group	**Type of messages returned or suppressed**
`c99|noc99`	C code that may behave differently between C89 and C99 language levels.
`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.
`inl|noinl`	Functions not inlined.
`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.
`vft|novft`	Generation of virtual function tables in C++ programs.
`zea|nozea`	Zero-extent arrays.

Notes

You can use the **#pragma info** directive to temporarily override the current **-qinfo** compiler option settings specified on the command line, in the configuration file, or by earlier invocations of the **#pragma info** directive.

Example

For example, in the code segments below, 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 invoked.

```
#pragma info(eff, nouni)
int MyFunction1()
{
  .
  .
  .

}

#pragma info(restore)
int MyFunction2()
{
  .
  .
  .

}
```

Related references

- "info" on page 119

#pragma instantiate

C++

Description

The **#pragma instantiate** directive instructs the compiler to immediately instantiate the specified template declaration.

Syntax

```
►►—#—pragma—instantiate—template————————————————————————————►◄
```

where *template* is a class template-id. For example:

```
#pragma instantiate Stack < int >
```

Notes

Use this pragma if you are migrating existing code. New code should use standard C++ explicit instantiation.

If you are handling template instantiations manually (that is, **-qnotempinc** and **-qnotemplateregistry** are specified), using **#pragma instantiate** will ensure that the specified template instantiation will appear in the compilation unit.

Related References

"General Purpose Pragmas" on page 259
"#pragma define" on page 268
"#pragma do_not_instantiate" on page 270
"tempinc" on page 233
"templateregistry" on page 235

#pragma ishome

C++

Description

The **#pragma ishome** directive 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──(──className──)────────────────────────────►◄
```

where:

className
: Is the literal name of the class whose home will be the current compilation unit.

Notes

A warning will be produced if there is a **#pragma ishome** without a matching **#pragma hashome**.

Example

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 References

"General Purpose Pragmas" on page 259
"#pragma hashome" on page 277

#pragma isolated_call

C C++

Description

The **#pragma isolated_call** directive marks a function that does not have or rely on side effects, other than those implied by its parameters.

Syntax

```
►►—#—pragma—isolated_call—(—function—)——————————————————————►◄
```

where *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. If the name refers to an overloaded function, all variants of that function are marked as isolated calls.

Notes

The **-qisolated_call** compiler option has the same effect as this pragma.

The pragma informs the compiler that the function listed does not have or rely on side effects, other than those implied by its parameters. Functions are considered to have or rely on side effects if they:

- Access a volatile object
- Modify an external object
- Modify a static object
- Modify a file
- Access a file that is modified by another process or thread
- Allocate a dynamic object, unless it is released before returning
- Release a dynamic object, unless it was allocated during the same invocation
- Change system state, such as rounding mode or exception handling
- Call a function that does any of the above

Essentially, any change in the state of the runtime environment is considered a side effect. Modifying function arguments passed by pointer or by reference is the only side effect that is allowed. Functions with other side effects can give incorrect results when listed in **#pragma isolated_call** directives.

Marking a function as **isolated_call** 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.

The function specified is permitted to examine non-volatile external objects and return a result that depends on the non-volatile state of the runtime environment. The function can also modify the storage pointed to by any pointer arguments passed to the function, that is, calls by reference. Do not specify a function that calls itself or relies on local static storage. Listing such functions in the **#pragma isolated_call** directive can give unpredictable results.

The **-qignprag** compiler option causes aliasing pragmas to be ignored. Use the **-qignprag** compiler option to debug applications containing the **#pragma isolated_call** directive.

Example

The following example shows the use of the **#pragma isolated_call** directive. Because the function **this_function** does not have side effects, a call to it will not change the value of the external variable **a**. The compiler can assume that the argument to **other_function** has the value 6 and will not reload the variable from memory.

```
int a;

// Assumed to have no side effects
int this_function(int);

#pragma isolated_call(this_function)
that_function()
{
   a = 6;
   // Call does not change the value of "a"
   this_function(7);

   // Argument "a" has the value 6
   other_function(a);
}
```

Related References

"General Purpose Pragmas" on page 259
"ignprag" on page 118
"isolated_call" on page 138

#pragma langlvl

▶ C

Description

The **#pragma langlvl** directive selects the C language level for compilation.

Syntax

```
▶▶—#—pragma—langlvl—(——language——)——————————————————————————▶◀
```

where values for *language* are described below.

▶ C For C programs, you can specify one of the following values for *language*:

`classic`	Allows the compilation of non-stdc89 programs, and conforms closely to the K&R level preprocessor.
`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 to the ANSI C89 standard, also known as ISO C90.
`stdc99`	Compilation conforms to the ISO C99 standard.
`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.

Default

The default language level varies according to the command you use to invoke the compiler:

Invocation	**Default language level**
xlc	extc89
cc	extended
c89	stdc89
c99	stdc99

Notes

This pragma can be specified only once in a source file, and it must appear before any noncommentary statements in a source file.

The compiler uses predefined macros in the header files to make declarations and definitions available that define the specified language level.

This directive can dynamically alter preprocessor behavior. As a result, compiling with the **-E** compiler option may produce results different from those produced when not compiling with the **-E** option.

Related References

"General Purpose Pragmas" on page 259
"E" on page 88
"langlvl" on page 143

See also the *IBM C Language Extensions* and *IBM C++ Language Extensions* sections of the *C/C++ Language Reference*.

#pragma leaves

C C++

Description

The **#pragma leaves** directive takes a function name and specifies that the function never returns to the instruction after the call.

Syntax

```
             ┌─,─────────┐
             ▼           │
►►─#─pragma─leaves─(───function─┴─)─────────────────────────►◄
```

Notes

This pragma tells the compiler that *function* never returns to the caller.

The advantage of the pragma is that it allows the compiler to ignore any code that exists after *function*, in turn, the optimizer can generate more efficient code. This pragma is commonly used for custom error-handling functions, in which programs can be terminated if a certain error is encountered. Some functions which also behave similarily are **exit**, **longjmp**, and **terminate**.

Example

```
#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 References

"General Purpose Pragmas" on page 259

#pragma loop_id

▶ C ▶ C++

Description

Marks a block with a scope-unique identifier.

Syntax

```
▶▶—#—pragma—loopid—(—name—)————————————————————————————————▶◀
```

where *name* is an identifier that is unique within the scoping unit.

Notes

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.

Related References

"General Purpose Pragmas" on page 259
"unroll" on page 244
"#pragma block_loop" on page 264
"#pragma unroll" on page 317
"#pragma unrollandfuse" on page 319

#pragma map

C C++

Description

The **#pragma map** directive tells the compiler that all references to an identifier are to be converted to *"name"*.

Syntax

```
►►—#—pragma—map—(—┬—identifier————————┬—,—"name"—)——————————————————►◄
                  └—function_signature—┘
```

where:

identifier
: A name of a data object or a nonoverloaded function with external linkage.
C++ If the identifier is the name of an overloaded function or a member function, there is a risk that the pragma will override the compiler-generated names. This will create problems during linking.

function_signature
: A name of a function or operator with internal linkage. The name can be qualified.

name
: The external name that is to be bound to the given object, function, or operator.
C++ Specify the mangled name if linking into a C++ name (a name that will have C++ linkage signature, which is the default signature in C++). See Example 4, in the **Examples** section below.

Notes

You should not use **#pragma map** to map the following:
- C++ Member functions
- Overloaded functions
- Objects generated from templates
- Functions with built in linkage

The directive can appear anywhere in the program. The identifiers appearing in the directive, including any type names used in the prototype argument list, are resolved as though the directive had appeared at file scope, independent of its actual point of occurrence.

Examples

Example 1 C

```
int funcname1()
{
    return 1;
}

#pragma map(func , "funcname1")    /* maps func to funcname1 */

int main()
{
   return func();       /* no function prototype needed in C */
}
```

Example 2 C++

```
extern "C" int funcname1()
{
  return 0;
}

extern "C" int func();  //function prototypes needed in C++

#pragma map(func , "funcname1") // maps ::func to funcname1

int main()
{
  return func();
}
```

Example 3 C++

```
#pragma map(foo, "bar")

int foo();             //function prototypes needed in C++

int main()
{
   return foo();
}

extern "C" int bar() {return 7;}
```

Related References

"General Purpose Pragmas" on page 259

#pragma mc_func

C C++

Description

The **#pragma mc_func** directive lets you define a function containing a short sequence of machine instructions.

Syntax

```
                                      ┌──────────────────┐
                                      ▼                  │
►►──#──pragma──mc_func──function──{───instruction_seq───┴──}──────────────────►◄
```

where:

function	Should specify a previously-defined function in a C or C++ program. If the function is not previously-defined, the compiler will treat the pragma as a function definition.
instruction_seq	Is a string containing a sequence of zero or more hexadecimal digits. The number of digits must comprise an integral multiple of 32 bits.

Notes

The **mc_func** pragma lets you 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 the **asm** keyword found in this and other compilers.

The **mc_func** pragma defines a function and should appear in your program source only where functions are ordinarily defined. The function name defined by **#pragma mc_func** should be previously declared or prototyped.

The compiler passes parameters to the function in the same way as 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. See **#pragma reg_killed_by** for a list of volatile registers available on your system.

Code generated from *instruction_seq* 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.

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**.

Example

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 ints with so-called *end-around carry*; that is, if a carry out results from the add then add the carry to the sum. This is frequently used in checksum computations.

The example also shows the use of **#pragma reg_killed_by** to list a specific set of volatile registers that can be altered by the function defined by **#pragma mc_func**.

```
int add_logical(int, int);
#pragma mc_func add_logical {"7c632014" "7c630194"}
                /*   addc       r3 <- r3, r4           */
                /*   addze      r3 <- r3, carry bit    */

#pragma reg_killed_by add_logical gr3, xer
                /* only gpr3 and the xer are altered by this function */


main() {

      int i,j,k;

      i = 4;
      k = -4;
      j = add_logical(i,k);
      printf("\n\nresult = %d\n\n",j);
}
```

Related References

"General Purpose Pragmas" on page 259
"#pragma isolated_call" on page 287
"#pragma reg_killed_by" on page 310
"asm" on page 57

#pragma nosimd

C C++

Description

The **#pragma nosimd** directive instructs the compiler to *not* generate VMX (Vector Multimedia Extension) instructions in the **for** loop immediately following this directive.

Syntax

```
►►──#──pragma──nosimd──────────────────────────────────────────────►◄
```

Notes

This directive has effect only when **-qhot=simd** is in effect together with **-qarch=ppc970**. With these compiler options in effect, the compiler will convert certain operations that are performed in a loop on successive elements of an array into a call to VMX (Vector Multimedia Extension) instruction. This call calculates several results at one time, which is faster than calculating each result sequentially.

The **#pragma nosimd** directive prevents the compiler from generating VMX (Vector Multimedia Extension) instructions for a specific **for** loop. The **for** loop affected by this directive must be the first program statement following the directive.

The **#pragma nosimd** directive applies only to the **for** loop immediately following it. The directive has no effect on other **for** loops that may be nested within the specified loop.

You can use **#pragma nosimd** together with loop optimization and parallelization directives.

Related References

"General Purpose Pragmas" on page 259
"arch" on page 56
"enablevmx" on page 92
"hot" on page 113

#pragma novector

▶ C ▶ C++

Description

The **#pragma novector** directive instructs the compiler to *not* auto-vectorize the next loop.

Syntax

```
▶▶—#—pragma—novector——————————————————————————————————————————▶◀
```

Notes

This directive has effect only when **-qhot=vector** is in effect. With **-qhot=vector** in effect, the compiler will 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 vector library routine. This call will calculate several results at one time, which is faster than calculating each result sequentially.

The **#pragma novector** directive prevents the compiler from auto-vectorizing instructions for a specific **for** loop. The **for** loop affected by this directive must be the first program statement following the directive.

The **#pragma novector** directive applies only to the **for** loop immediately following it. The directive has no effect on other **for** loops that may be nested within the specified loop.

You can use **#pragma novector** together with loop optimization and parallelization directives.

Related References

"General Purpose Pragmas" on page 259
"hot" on page 113

#pragma options

C C++

Description

The **#pragma options** directive specifies compiler options for your source program.

Syntax

```
                                 .------------------------------------.
                                 V                                    |
>>--#--pragma--+-option--+--------+-option_keyword--------------------+--+---------------><
               '-options-'        |                  .-;-------.    |
                                  |                  | .-,---. |    |
                                  |                  V V     | |    |
                                  '-option_keyword--=----value-+-+----'
```

Notes

By default, pragma options generally apply to the entire compilation unit.

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
```

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 ... */
```

Options specified before any code in your source program apply to your entire compilation unit. You can use other **#pragma** directives throughout your program to turn an option on for a selected block of source code. For example, you can request that 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
```

The settings in the table below are valid *options* for **#pragma options**. For more information, refer to the pages of the equivalent compiler option.

Language Application		Valid settings for #pragma options *option_keyword*	Compiler option equivalent	Description
C	C++	align=*option*	-qalign	Specifies what aggregate alignment rules the compiler uses for file compilation.
C	C++	[no]ansialias	-qalias	Specifies whether type-based aliasing is to be used during optimization.
C	C++	assert=*option*	-qalias	Requests the compiler to apply aliasing assertions to your compilation unit.
C	C++	[no]attr attr=full	-qattr	Produces an attribute listing containing all names.
C	C++	chars=*option*	-qchars See also #pragma chars	Instructs the compiler to treat all variables of type char as either signed or unsigned.
C	C++	[no]check	-qcheck	Generates code which performs certain types of run-time checking.
C	C++	[no]compact	-qcompact	When used with optimization, reduces code size where possible, at the expense of execution speed.
C	C++	[no]dbcs	-qmbcs, dbcs	String literals and comments can contain multibyte characters.
C		[no]dbxextra	-qdbxextra	Generates symbol table information for unreferenced variables.
C	C++	[no]digraph	-qdigraph	Allows special digraph and keyword operators.
C	C++	[no]dollar	-qdollar	Allows the $ symbol to be used in the names of identifiers.
C	C++	enum=*option*	-qenum See also #pragma enum	Specifies the amount of storage occupied by the enumerations.
C	C++	flag=*option*	-qflag	Specifies the minimum severity level of diagnostic messages to be reported. Severity levels can also be specified with: #pragma options flag=i => #pragma report (level,I) #pragma options flag=w => #pragma report (level,W) #pragma options flag=e,s,u => #pragma report (level,E)
C	C++	float=[no]*option*	-qfloat	Specifies various floating point options to speed up or improve the accuracy of floating point operations.
C	C++	[no]flttrap=*option*	-qflttrap	Generates extra instructions to detect and trap floating point exceptions.
C	C++	[no]fullpath	-qfullpath	Specifies the path information stored for files for dbx stabstrings.

Language Application		Valid settings for #pragma options *option_keyword*	Compiler option equivalent	Description
C	C++	[no]funcsect	-qfuncsect	Places intructions for each function in a separate cset.
C	C++	halt	-qhalt	Stops compiler when errors of the specified severity detected.
C	C++	[no]idirfirst	-qidirfirst	Specifies search order for user include files.
C	C++	[no]ignerrno	-qignerrno	Allows the compiler to perform optimizations that assume **errno** is not modified by system calls.
C	C++	ignprag=*option*	-qignprag	Instructs the compiler to ignore certain pragma statements.
C	C++	[no]info=*option*	-qinfo See also #pragma info	Produces informational messages.
C	C++	initauto=*value*	-qinitauto	Initializes automatic storage to a specified hexadecimal byte value.
C	C++	[no]inlglue	-qinlglue	Generates fast external linkage by inlining the pointer glue code necessary to make a call to an external function or a call through a function pointer.
C	C++	isolated_call=*names*	-qisolated_call See also #pragma isolated_call	Specifies functions in the source file that have no side effects.
C		langlvl	-qlanglvl	Specifies different language levels. This directive can dynamically alter preprocessor behavior. As a result, compiling with the **-E** compiler option may produce results different from those produced when not compiling with the **-E** option.
C	C++	[no]libansi	-qlibansi	Assumes that all functions with the name of an ANSI C library function are in fact the system functions.
C	C++	[no]list	-qlist	Produces a compiler listing that includes an object listing.
C	C++	[no]longlong	-qlonglong	Allows long long types in your program.
C	C++	[no]maxmem=*number*	-qmaxmem	Instructs the compiler to halt compilation when a specified number of errors of specified or greater severity is reached.
C	C++	[no]mbcs	-qmbcs, dbcs	String literals and comments can contain multibyte characters.

Language Application		Valid settings for #pragma options *option_keyword*	Compiler option equivalent	Description
C	C++	[no]optimize optimize=*number*	-O, optimize	Specifies the optimization level to apply to a section of program code. The compiler will accept the following values for *number*: • **0** - sets level 0 optimization • **2** - sets level 2 optimization • **3** - sets level 3 optimization If no value is specified for *number*, the compiler assumes level 2 optimization.
	C++	priority=*number*	-qpriority See also "#pragma priority" on page 308	Specifies the priority level for the initialization of static constructors
C	C++	[no]proclocal, [no]procimported, [no]procunknown	-qproclocal, procimported, procunknown	Marks functions as local, imported, or unknown.
C		[no]proto	-qproto	If this option is set, the compiler assumes that all functions are prototyped.
C	C++	[no]ro	-qro	Specifies the storage type for string literals.
C	C++	[no]roconst	-qroconst	Specifies the storage location for constant values.
C	C++	[no]showinc	-qshowinc	If used with **-qsource**, all include files are included in the source listing.
C	C++	[no]source	-qsource	Produces a source listing.
C	C++	spill=*number*	-qspill	Specifies the size of the register allocation spill area.
C	C++	[no]stdinc	-qstdinc	Specifies which files are included with #include <file_name> and #include ″file_name″ directives.
C	C++	[no]strict	-qstrict	Turns off aggressive optimizations of the **-O3** compiler option that have the potential to alter the semantics of your program.
C	C++	tbtable=*option*	-qtbtable	Changes the length of tabs as perceived by the compiler.
C	C++	tune=*option*	-qtune	Specifies the architecture for which the executable program is optimized.
C	C++	[no]unroll unroll=*number*	-qunroll	Unrolls inner loops in the program by a specified factor.
C		[no]upconv	-qupconv	Preserves the unsigned specification when performing integral promotions.
	C++	[no]vftable	-qvftable	Controls the generation of virtual function tables.

Language Application		Valid settings for #pragma options *option_keyword*	Compiler option equivalent	Description
C	C++	[no]xref	-qxref	Produces a compiler listing that includes a cross-reference listing of all identifiers.

Related References

"General Purpose Pragmas" on page 259
"E" on page 88

#pragma option_override

C C++

Description

The **#pragma option_override** directive lets you specify alternate optimization options to apply to specific functions.

Syntax

```
                                          ┌─,───────────────┐
                                          ▼                 │
►►─#─pragma─option_override─(─fname───"─option─"─┴─)───────────────►◄
```

Valid settings and syntax for *option*, and their corresponding command line options, are shown below:

Settings and Syntax for #pragma option_override *option*	**Command Line Option**	**Examples**
opt(level,*number*)	-O, -O2, -O3, -O4, -O5	#pragma option_override (*fname*, "opt(level, 3)")
opt(registerSpillSize,*num*)	-qspill=*num*	#pragma option_override (*fname*, "opt(registerSpillSize,512)")
opt(size[,yes])	-qcompact	#pragma option_override (*fname*, "opt(size)") #pragma option_override (*fname*, "opt(size,yes)")
opt(size,no)	-qnocompact	#pragma option_override (*fname*, "opt(size,no)")
opt(strict)	-qstrict	#pragma option_override (*fname*, "opt(strict)")
opt(strict,no)	-qnostrict	#pragma option_override (*fname*, "opt(strict,no)")

Notes

By default, optimization options specified on the command line apply to the entire source program. However, certain types of runtime errors may occur only when optimization is turned on. This pragma lets you override command line optimization settings for specific functions (*fname*) in your program, which may be useful in identifying and correcting programming errors in those functions.

Per-function optimizations have effect only if optimization is already enabled by compilation option. You can request per-function optimizations at a level less than that applied to the rest of the program being compiled. Selecting options through this pragma affects only the specific optimization option selected, and does not affect the implied settings of related options.

Options are specified in double quotes, so they are not subject to macro expansion. The option specified within quotes must comply with the syntax of the build option.

This pragma cannot be used with overloaded member functions.

This pragma affects only functions defined in your compilation unit and can appear anywhere in the compilation unit, for example:

- before or after a compilation unit
- before or after the function definition
- before or after the function declaration
- before or after a function has been referenced

- inside or outside a function definition.

Related References

"General Purpose Pragmas" on page 259
"compact" on page 72
"O, optimize" on page 179
"spill" on page 219
"strict" on page 225

#pragma pack

C C++

Description

The **#pragma pack** directive modifies the current alignment rule for members of structures following the directive.

Syntax

```
►►──#──pragma──pack──(─┬────────┬─)──────────────────────────────────────►◄
                       ├─nopack─┤
                       ├─1──────┤
                       ├─2──────┤
                       ├─4──────┤
                       ├─8──────┤
                       ├─16─────┤
                       └─pop────┘
```

where:

1 | 2 | 4 | 8 | 16	Members of structures are aligned on the specified byte-alignment, or on their natural alignment boundary, whichever is less, and the specified value is pushed on the stack.
nopack	No packing is applied, and ″nopack″ is pushed onto the pack stack
pop	The top element on the pragma pack stack is popped.
(*no argument specified*)	Specifying #pragma pack() has the same effect as specifying #pragma pack(pop).

Notes

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 according to the alignment rule.

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 integer (int), a **#pragma pack(2)** directive would cause that member to be packed in the structure on a 2-byte boundary, while a **#pragma pack(4)** directive would have no effect.

The **#pragma pack** directive is stack based. All pack values are pushed onto a stack as the source code is parsed. The value at the top of the current pragma pack stack is the value used to pack members of all subsequent structures within the scope of the current alignment rule.

A **#pragma pack** stack is associated with the current element in the alignment rule stack. Alignment rules are specified with the **-qalign** compiler option or with the **#pragma options align** directive. If a new alignment rule is created, a new **#pragma pack** stack is created. If the current alignment rule is popped off the alignment rule stack, the current **#pragma pack** stack is emptied and the previous **#pragma pack** stack is restored. Stack operations (pushing and popping pack settings) affect only the current **#pragma pack** stack.

The **#pragma pack** directive causes bit fields to cross bit field container boundaries.

Examples

1. In the code shown below, the structure s_t2 will have its members packed to 1-byte, but structure s_t1 will not be affected. This is because the declaration for s_t1 began before the pragma directive. However, s_t2 is affected because its declaration began after the pragma directive.

```
struct s_t1 {
    char a;
    int b;
    #pragma pack(1)
    struct s_t2 {
            char x;
            int y;
    } S2;
    char c;
    int d;
} S1;
```

2. 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):**
sizeof s_t = 16	sizeof s_t = 11
offsetof a = 0	offsetof a = 0
offsetof b = 4	offsetof b = 1
offsetof c = 8	offsetof c = 5
offsetof d = 12	offsetof d = 7
align of a = 1	align of a = 1
align of b = 4	align of b = 1
align of c = 2	align of c = 1
align of d = 4	align of d = 1

Related References

"General Purpose Pragmas" on page 259
"align" on page 50
"#pragma options" on page 299

#pragma priority

C++

Description

The **#pragma priority** directive specifies the order in which static objects are to be initialized.

Syntax

```
►►—#—pragma—priority—(—n—)——————————————————————————————————►◄
```

Notes

The value of *n* must be an integer literal in the range of 101 to 65535. The default value is 65535. A lower value indicates a higher priority; a higher value indicates a lower priority.

The priority value applies to all global and static objects following the **#pragma priority** directive, unless an explicit value is given by the variable attribute init_priority or another **#pragma priority** directive is encountered.

Objects with the same priority value are constructed in declaration order. Use **#pragma priority** to specify the construction order of objects across files. However, if you are creating an executable or shared library target from source files, the compiler will check dependency ordering, which may override **#pragma priority**.

For example, if a copy of object A is passed as a parameter to the object B constructor, then the compiler will arrange for A to be constructed first, even if this violates the top-to-bottom or **#pragma priority** ordering. This is essential for orderless programming, which the compiler permits. If the target is an .obj/.lib, this processing is not done, because there may not be enough information to detect the dependencies.

Variable attribute init_priority: The C++ variable attribute init_priority can also be used to assign a priority level to a shared variables of class type. See *XL C/C++ Language Reference* for more information.

Example

```
#pragma priority(1001)
```

Related references

- "info" on page 119

#pragma reachable

C C++

Description

The **#pragma reachable** directive declares that the point after the call to a routine, *function*, can be the target of a branch from some unknown location. This pragma should be used in conjunction with the `setjmp` macro.

Syntax

```
                                   ┌─,───────┐
                                   ▼         │
►►──#──pragma──reachable──(────function──┴──)───────────────────────────►◄
```

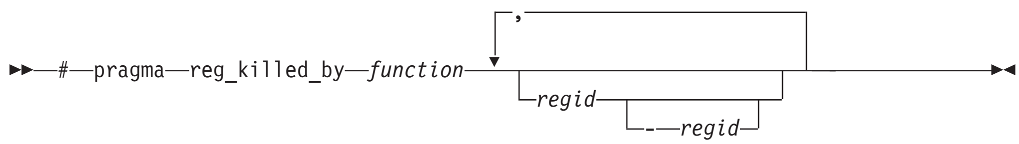
#pragma reg_killed_by

C C++

Description

The **#pragma reg_killed_by** directive specifies a set of volatile registers that may be altered (killed) by the specified function. This pragma can only be used on functions that are defined using **#pragma mc_func**.

Syntax



where:

function The function previously defined using the **#pragma mc_func**.

regid The symbolic name(s) of either a single register or a range of registers to be altered by the named *function*. A range of registers is identified by providing the symbolic names of both starting and ending registers, separated by a dash. If no registers are specified, no registers will be altered by the specified *function*.

The symbolic name is made up of two parts. The first part is the register class name, specified using a sequence of one or more characters in the range of "a" to "z" and/or "A" to "Z".

The second part is a integral number in the range of unsigned int. This number identifies a specific register number within a register class. Some register classes do not require that a register number be specified, and an error will result if you try to do so.

If *regid* is not specified, no volatile registers will be killed by the named *function*.

Registers	
Class and [Register numbers]	**Description and usage**
ctr	Count register (CTR)
cr[0-7]	Condition register (CR) • Each register in this class is one of the 4-bit fields in the condition register. • Of the 8 CR fields, only **cr0**, **cr1**, and **cr5-cr7** can be specified by **#pragma reg_killed_by**.
fp[0-31]	Floating point registers (FPR) • Of the 32 machine registers, only **fp0-fp13** can be specified by **#pragma reg_killed_by**.
fs	Floating point status and control register (FPSCR)
lr	Link register (LR)
gr[0-31]	General purpose registers (GPR) • Of the 32 machine registers, only **gr0** and **gr3-gr12** can be specified by **#pragma reg_killed_by**.
vr[0–31]	Vector registers (Altivec processors only)
xer	Fixed point exception (XER)

Notes

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.

Registers specified by *regid* must meet the following requirements:

- the class name part of the register name must be valid
- the register number is either required or prohibited
- when the register number is required, it must be in the valid range

If any of these requirements are not met, an error is issued and the pragma is ignored.

Example

The following example shows how to use **#pragma reg_killed_by** to list a specific set of volatile registers to be used by the function defined by **#pragma mc_func**.

```
int add_logical(int, int);
#pragma mc_func add_logical {"7c632014" "7c630194"}
                /*   addc       r3 <- r3, r4           */
                /*   addze      r3 <- r3, carry bit    */

#pragma reg_killed_by add_logical gr3, xer
                /* only gpr3 and the xer are altered by this function */


main() {

      int i,j,k;

      i = 4;
      k = -4;
      j = add_logical(i,k);
      printf("\n\nresult = %d\n\n",j);
}
```

Related References

"General Purpose Pragmas" on page 259
"#pragma mc_func" on page 295

#pragma report

C++

Description

The **#pragma report** directive controls the generation of specific messages. The pragma will take precedence over **#pragma info**. Specifying **#pragma report(pop)** will revert the report level to the previous level. If no previous report level was specified, then a warning will be issued and the report level will remain unchanged.

Syntax

```
>>--#--pragma--report--(--+-level--,--+-E-+------------------+--)-------------><
                          |           +-W-+                  |
                          |           '-I-'                  |
                          +-+-enable--+--,--message_number---+
                          | '-disable-'                      |
                          '-pop------------------------------'
```

where:

level	Indicates the minimum severity level of diagnostic messages to display.
E | W | I	Used in conjunction with **level** to determine the type of diagnostic messages to display.

E Signifies a minimum message severity of ’error’. This is considered as the most severe type of diagnostic message. A report level of ’E’ will display only ’error’ messages. An alternative way of setting the report level to ’E’ is by specifying the **-qflag=e:e** compiler option.

W Signifies a minimum message severity of ’warning’. A report level of ’W’ will filter out all informational messages, and display only those messages classified as warning or error messages. An alternative way of setting the report level to ’W’ is by specifying the **-qflag=w:w** compiler option.

I Signifies a minimum message severity of ’information’. Information messages are considered as the least severe type of diagnostic message. A level of ’I’ would display messages of all types. The compiler sets this as the default option. An alternative way of setting the report level to ’I’ is by specifying the **-qflag=i:i** compiler option.

enable | disable	Enables or disables the specified message number.
message_number	Is an identifier containing the message number prefix, followed by the message number. An example of a message number is: CPPC1004
pop	resets the report level back to the previous report level. If a pop operation is performed on an empty stack, the report level will remain unchanged and no message will be generated.

Examples

1. Specifying **#pragma info** instructs the compiler to print all informational diagnostics. The pragma report instructs the compiler to display only those messages with a severity of ’W’ or warning messages. In this case, none of the informational diagnostics will be displayed.

```
#pragma info(all)
#pragma report(level, W)
```

2. If CPPC1000 was an error message, it would be displayed. If it was any other type of diagnostic message, it would not be displayed.

   ```
   #pragma report(enable, CPPC1000)  // enables message number CPPC1000
   #pragma report(level, E)   // display only error messages.
   ```

   Changing the order of the code like so:

   ```
   #pragma report(level, E)
   #pragma report(enable, CPPC1000)
   ```

   would yield the same result. The order in which the two lines of code appear in, does not affect the outcome. However, if the message was 'disabled', then regardless of what report level is set and order the lines of code appear in, the diagnostic message will not be displayed.
3. In line 1 of the example below, the initial report level is set to 'I', causing message CPPC1000 to display regardless of the type of diagnostic message it is classified as. In line 3, a new report level of 'E' is set, indicating only messages with a severity level of 'E' will be displayed. Immediately following line 3, the current level 'E' is 'popped' and reset back to 'I'.

   ```
   1  #pragma report(level, I)
   2  #pragma report(enable, CPPC1000)
   3  #pragma report(level, E)
   4  #pragma report(pop)
   ```

Related References

"General Purpose Pragmas" on page 259
"flag" on page 98

#pragma stream_unroll

C C++

Description

Breaks a stream contained in a **for** loop into multiple streams.

Syntax

```
>>─#─pragma─stream_unroll─(─┬───┬─)─────────────────────────────────────><
                            └─n─┘
```

where *n* is a loop unrolling factor. In C programs, the value of *n* is a positive integral constant expression. In C++ programs, the value of *n* is a positive scalar integer or compile-time constant initialization expression. An unroll factor of 1 disables unrolling. If *n* is not specified and if **-qhot**, **-qsmp**, or **-O4** or higher is specified, the optimizer determines an appropriate unrolling factor for each nested loop.

Notes

Neither **-O3** nor **-qipa=level=2** is sufficient to enable stream unrolling. You must additionally specify **-qhot** or **-qsmp**, or use optimization level **-O4** or higher.

For stream unrolling to occur, the **#pragma stream_unroll** directive must be the last pragma specified preceding a **for** loop. Specifying **#pragma stream_unroll** more than once for the same **for** loop or combining it with other loop unrolling pragmas (**unroll**, **nounroll**, **unrollandfuse**, **nounrollandfuse**) also results in a warning from XL C; XL C++ silently ignores all but the last of multiple loop unrolling pragmas specified on the same **for** loop.

Stream unrolling does not occur when **#pragma stream_unroll** is followed by **#pragma block_loop**. In this situation, XL C also emits a severe error; XL C++ emits no diagnostic, but the optimizer might not apply the stream unrolling optimization.

Stream unrolling is also suppressed by compilation under certain optimization options. If option **-qstrict** is in effect, no stream unrolling takes place. Therefore, if you want to enable stream unrolling with the **-qhot** option alone, you must also specify **-qnostrict**.

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];
}
```

The increased number of read and store operations are distributed among a number of streams determined by the compiler, reducing computation time and boosting performance.

Related references
- "unroll" on page 244
- "#pragma unroll" on page 317
- "#pragma unrollandfuse" on page 319

#pragma strings

C C++

Description

The **#pragma strings** directive sets the storage type for string literals and specifies whether they can be placed in read-only or read-write memory.

Syntax

```
>>--#--pragma--strings--(--+-writeable-+--)--------------------><
                           '-readonly--'
```

Notes

C Strings are read-only by default if any form of the compiler invocation **xlc** is used.

C++ Strings are read-only by default if any form of the compiler invocations **xlC** or **xlc++** is used.

This pragma must appear before any source statements in order to have effect.

Example

```
#pragma strings(writeable)
```

Related references

- "roconst" on page 206

#pragma unroll

C C++

Description

The **#pragma unroll** directive is used to unroll the innermost or outermost loops in your program, which can help improve program performance.

Syntax

```
►►─#─pragma─┬─nounroll──────────┬──────────────────────────────────►◄
            └─unroll─(─┬───┬─)─┘
                       └─n─┘
```

where *n* is the loop unrolling factor. In C programs, the value of *n* is a positive integral constant expression. In C++ programs, the value of *n* is a positive scalar integer or compile-time constant initialization expression. An unroll factor of 1 disables unrolling. If *n* is not specified and if **-qhot**, **-qsmp**, or **-O4** or higher is specified, the optimizer determines an appropriate unrolling factor for each nested loop.

Notes

The **#pragma unroll** and **#pragma nounroll** directives must appear immediately before the loop to be affected.

Only one of these directives can be specified for a given loop. 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.

Specifying **#pragma nounroll** for a loop instructs the compiler to not unroll that loop. Specifying **#pragma unroll(1)** has the same effect.

To see if the **unroll** option improves performance of a particular application, you should first compile the program with usual options, then run it with a representative workload. You should then recompile with command line **-qunroll** option and/or the **unroll** pragmas enabled, then rerun the program under the same conditions to see if performance improves.

Examples

1. In the following example, loop control is not modified:

   ```
   #pragma unroll(2)
   while (*s != 0)
   {
     *p++ = *s++;
   }
   ```

   Unrolling this by a factor of 2 gives:

```
while (*s)
{
  *p++ = *s++;
  if (*s == 0) break;
  *p++ = *s++;
}
```

2. In this example, loop control *is* modified:

```
#pragma unroll(3)
for (i=0; i<n; i++) {
  a[i]=b[i] * c[i];
}
```

Unrolling by 3 gives:

```
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 references

- "unroll" on page 244
- "#pragma unrollandfuse" on page 319

#pragma unrollandfuse

C C++

Description

This pragma instructs the compiler to attempt an unroll and fuse operation on nested **for** loops.

Syntax

```
>>—#—pragma—┬—nounrollandfuse———————————————┬——————————————————————————————————————————><
            └—unrollandfuse—(—┬———┬—)———————┘
                              └—n—┘
```

where *n* is a loop unrolling factor. In C programs, the value of *n* is a positive integral constant expression. In C++ programs, the value of *n* is a positive scalar integer or compile-time constant initialization expression. If *n* is not specified and if **-qhot**, **-qsmp**, or **-O4** or higher is specified, the optimizer determines an appropriate unrolling factor for each nested loop.

Notes

The **#pragma unrollandfuse** directive applies only to the outer loops of nested **for** loop structures 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 **block_loop**, **nounrollandfuse**, **nounroll**, **unroll**, or **stream_unroll** directives for the same **for** loop.

Specifying **#pragma nounrollandfuse** instructs the compiler to not unroll that loop.

Examples

1. 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 structure 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];
    }
}
```

2. 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 references

- "unroll" on page 244
- "#pragma unroll" on page 317

#pragma weak

▶ C ▶ C++

Description

The **#pragma weak** directive prevents the link editor from issuing error messages if it does not find a definition for a symbol, or if it encounters a symbol multiply-defined during linking.

Syntax

```
▸▸──#──pragma──weak──identifier──┬─────────────────┬──────────────────────────▸◂
                                  └─=──identifier2─┘
```

Notes

While this pragma is intended for use primarily with functions, it will also work for most data objects.

This pragma should not be used with uninitialized global data, or with shared library data objects that are exported to executables.

The dynamic linker will use the definition in whatever object appears first on the command line. Thus, the order in which the object files are presented to the linker is important.

Two forms of **#pragma weak** can be specified in your program source.

#pragma weak *identifier*
: This form of the pragma defines *identifier* as a weak global symbol. References to *identifier* uses the identifier value if it is defined, otherwise *identifier* is assigned a value of 0.

  If *Identifier* is defined in the same compilation unit as **#pragma weak** *identifier*, *identifier* is treated as a weak definition. If **#pragma weak** exists in a compilation unit that does not use or declare *identifier*, the pragma is accepted and ignored.

  If *identifier* denotes a function with C++ linkage, *identifier* must be specified using the C++ mangled name of the function. Also, if the C++ function is a template function, you must explicitly instantiate the template function.

#pragma weak *identifier***=***identifier2*
: This form of the pragma defines *identifier* as a weak global symbol. References to *identifier* will use the value of *identifier2*.

  Identifier2 must not be a member function.

  Identifier 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 *identifier* is declared in the compilation unit, *identifier*'s declaration must be compatible to that of *identifier2*. For example, if *identifier2* is a function, *identifier* must have the same return and argument types as *identifier2*.

  Identifier2 must be declared in the same compilation unit as **#pragma weak**.

If *identifier2* denotes a function with C++ linkage, the names of *identifier* and *identifier2* must be specified using the mangled names of the functions. If the C++ function is a template function, you must explicitly instantiate the template function.

The compiler will ignore **#pragma weak** and issue warning messages if:

- If *identifer2* (if specified) is not defined in the compilation unit.
- If *identifer2* (if specified) is a member function.
- If *identifer* is declared but its type is not compatible with that of *identifer2* (if specified).

The compiler will ignore **#pragma weak** and issue a severe error message if the weak *identifier* is defined.

Examples

1. The following is an example of the **#pragma weak** *identifier* form of the pragma:

   ```
   // Begin Compilation Unit 1
   #include <stdio.h>
   extern int foo;
   #pragma weak foo

   int main()
   {
      int *ptr;
      ptr = &foo;
      if (ptr == 0)
         printf("foo has been assigned a value of 0\n");
      else
         printf("foo was already defined\n");
   }
   //End Compilation Unit 1

   // Begin Compilation Unit 2
   int foo = 1;
   // End Compilation Unit 2
   ```

   If only Compilation Unit 1 is compiled to produce an executable, identifier `foo` will be defined and assigned the value `0`. The output from execution will be the string: ″`foo has been assigned a value of 0`.″
2. The following is an example of the **#pragma weak** *identifier*=*identifier2* form of the pragma:

   ```
   //Begin Compilation Unit
   extern "C" void printf(char *,...);

   void foo1(void)
   {
      printf("Just in function foo1()\n");
   }


   #pragma weak _Z3foov = _Z4foo1v

   int main()
   {
      foo();
   }
   //End Compilation Unit
   ```

Pragmas to Control Parallel Processing

The #pragma directives on this page give you control over how the compiler handles parallel processing in your program.

Directives apply only to the statement or statement block immediately following the directive.

OpenMP Pragma Directives C C++	Description
#pragma omp atomic	Identifies a specific memory location that must be updated atomically and not be exposed to multiple, simultaneous writing threads.
#pragma omp parallel	Defines a parallel region to be run by multiple threads in parallel. With specific exceptions, all other OpenMP directives work within parallelized regions defined by this directive.
#pragma omp for	Work-sharing construct identifying an iterative for-loop whose iterations should be run in parallel.
#pragma omp parallel for	Shortcut combination of **omp parallel** and **omp for** pragma directives, used to define a parallel region containing a single **for** directive.
#pragma omp ordered	Work-sharing construct identifying a structured block of code that must be executed in sequential order.
#pragma omp section, #pragma omp sections	Work-sharing construct identifying a non-iterative section of code containing one or more subsections of code that should be run in parallel.
#pragma omp parallel sections	Shortcut combination of **omp parallel** and **omp sections** pragma directives, used to define a parallel region containing a single **sections** directive.
#pragma omp single	Work-sharing construct identifying a section of code that must be run by a single available thread.
#pragma omp master	Synchronization construct identifying a section of code that must be run only by the master thread.
#pragma omp critical	Synchronization construct identifying a statement block that must be executed by a single thread at a time.
#pragma omp barrier	Synchronizes all the threads in a parallel region.
#pragma omp flush	Synchronization construct identifying a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory.
#pragma omp threadprivate	Defines the scope of selected file-scope data variables as being private to a thread, but file-scope visible within that thread.

Related Concepts

"Program Parallelization" on page 11

Related Tasks

"Control Parallel Processing with Pragmas" on page 31

Related References

"OpenMP Run-time Options for Parallel Processing" on page 352
"Built-in Functions Used for Parallel Processing" on page 354

For complete information about the OpenMP Specification, see:

- OpenMP Web site
- OpenMP Specification

#pragma omp atomic

C C++

Description

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_block>
```

where *statement* is an expression statement of scalar type that takes one of the forms that follow:

<table>
<tr><th>statement</th><th>Conditions</th></tr>
<tr><td>x bin_op = expr</td><td>where:
bin_op is one of:
<code>+ * - / & ^ | << >></code>
expr is an expression of scalar type that does not reference x.</td></tr>
<tr><td>x++</td><td></td></tr>
<tr><td>++x</td><td></td></tr>
<tr><td>x--</td><td></td></tr>
<tr><td>--x</td><td></td></tr>
</table>

Notes

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;
```

Related References

"Pragmas to Control Parallel Processing" on page 323

#pragma omp parallel

Description

The **omp parallel** directive explicitly instructs the compiler to parallelize the chosen segment of code.

Syntax

```
#pragma omp parallel [clause[[,] clause] ...]
<statement_block>
```

where *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 non-zero 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 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.

Notes

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.

Related References

"Pragmas to Control Parallel Processing" on page 323
"#pragma omp for" on page 328
"#pragma omp parallel for" on page 333
"#pragma omp parallel sections" on page 336

#pragma omp for

Description

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[[,] clause] ...]
<for_loop>
```

where *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 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: **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*. **static,1** Iterations of a loop are divided into chunks of size 1. Each chunk is assigned to a thread in *round-robin* fashion. This scheduling policy is also known as *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.

Notes

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 References

"Pragmas to Control Parallel Processing" on page 323
"#pragma omp parallel for" on page 333

#pragma omp ordered

Description

The **omp ordered** directive identifies a structured block of code that must be executed in sequential order.

Syntax

```
#pragma omp ordered
    statement_block
```

Notes

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.

Related References

"Pragmas to Control Parallel Processing" on page 323
"#pragma omp for" on page 328
"#pragma omp parallel for" on page 333

#pragma omp parallel for

Description

The **omp parallel for** directive effectively combines the **omp parallel** and **omp for** directives. This directive lets you define a parallel region containing a single **for** directive in one step.

Syntax

```
#pragma omp parallel for [clause[[,] clause] ...]
<for_loop>
```

Notes

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.

Related References

"Pragmas to Control Parallel Processing" on page 323
"#pragma omp for" on page 328
"#pragma omp parallel" on page 326

#pragma omp section, #pragma omp sections

Description

The **omp sections** directive distributes work among threads bound to a defined parallel region.

Syntax

```
#pragma omp sections [clause[ clause] ...]
   {
     [#pragma omp section]
          statement-block
     [#pragma omp section]
          statement-block
       .
       .
       .
   }
```

where *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.
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.
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.

Notes

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.

Related References

"Pragmas to Control Parallel Processing" on page 323
"#pragma omp parallel sections" on page 336

#pragma omp parallel sections

Description

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[[,] clause] ...]
   {
     [#pragma omp section]
          statement-block
     [#pragma omp section]
          statement-block
       .
       .
       .
     ]
   }
```

Notes

All clauses and restrictions described in the **omp parallel** and **omp sections** directives apply to the **omp parallel sections** directive.

Related References

"Pragmas to Control Parallel Processing" on page 323
"#pragma omp parallel" on page 326
"#pragma omp section, #pragma omp sections" on page 334

#pragma omp single

Description

The **omp single** directive identifies a section of code that must be run by a single available thread.

Syntax

```
#pragma omp single [clause[[,] clause] ...]
   statement_block
```

where *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.

Notes

An implied barrier exists at the end of a parallelized statement block unless the **nowait** clause is specified.

Related References

"Pragmas to Control Parallel Processing" on page 323

#pragma omp master

Description

The **omp master** directive identifies a section of code that must be run only by the master thread.

Syntax

```
#pragma omp master
    statement_block
```

Notes

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.

Related References

"Pragmas to Control Parallel Processing" on page 323

#pragma omp critical

Description

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)]
   statement_block
```

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.

Notes

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.

Related References

"Pragmas to Control Parallel Processing" on page 323

#pragma omp barrier

Description

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
```

Notes

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  */
```

Related References

"Pragmas to Control Parallel Processing" on page 323

#pragma omp flush

Description

The **omp flush** directive identifies a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory.

Syntax

```
#pragma omp flush [ (list) ]
```

where *list* is a comma-separated list of variables that will be synchronized.

Notes

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 conjuction 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  */
```

Related References

"Pragmas to Control Parallel Processing" on page 323
"#pragma omp barrier" on page 340
"#pragma omp critical" on page 339
"#pragma omp for" on page 328
"#pragma omp parallel" on page 326
"#pragma omp parallel for" on page 333
"#pragma omp parallel sections" on page 336
"#pragma omp section, #pragma omp sections" on page 334
"#pragma omp single" on page 337

#pragma omp threadprivate

Description

The **omp threadprivate** directive makes the named file-scope, namespace-scope, or static block-scope variables private to a thread.

Syntax

```
#pragma omp threadprivate (list)
```

where *list* is a comma-separated list of variables.

Notes

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.

Related References

"Pragmas to Control Parallel Processing" on page 323

Acceptable compiler mode and processor architecture combinations

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.

All PowerPC machines share a common set of instructions, but may also include additional instructions unique to a given processor or processor family.

The table below shows some selected processors, and the various features they may or may not support:

Processor	graphics support	sqrt support	64-bit support
rs64b	yes	yes	yes
rs64c	yes	yes	yes
pwr3	yes	yes	yes
pwr4	yes	yes	yes
pwr5	yes	yes	yes

If you want to generate code that will run across a variety of processors, use the following guidelines to select the appropriate **-qarch** and/or **-qtune** compiler options. Code compiled with:

- **-qarch=pwr4** will run only on POWER4 machines.
- **-qarch=pwr5** will run only on POWER5 machines.
- **-qarch=ppc** will run on any PowerPC system.
- **-q64** will run only on PowerPC machines with 64-bit support
- Other **-qarch** options that refer to specific processors will run on any functionally equivalent PowerPC machine. For example, the table that follows shows that code compiled with **-qarch=pwr3** will also run on a **rs64c**.

If you want to generate code optimized specifically for a particular processor, acceptable combinations of **-q32**, **-q64**, **-qarch**, and **-qtune** compiler options are shown in the following table.

Acceptable -qarch /-qtune combinations			
-qarch option	**Predefined macros**	**Default -qtune setting**	**Available -qtune settings**
ppc64v	_ARCH_PPCV	ppc970	auto ppc970

Related references

- "Specify Compiler Options for Architecture-Specific, 32- or 64-bit Compilation" on page 25
- "32, 64" on page 45
- "arch" on page 56
- "tune" on page 242

Compiler Messages

This section outlines some of the basic reporting mechanisms the compiler uses to describe compilation errors.

- "Message Severity Levels and Compiler Response"
- "Compiler Return Codes" on page 346
- "Compiler Message Format" on page 346

Message Severity Levels and Compiler Response

The following table shows the compiler response associated with each level of message severity.

Letter	Severity	Compiler Response
I	Informational	Compilation continues. The message reports conditions found during compilation.
W	Warning	Compilation continues. The message reports questionable and possibly unintended conditions. The program will run as written.
E	Error	C Compilation continues and object code is generated. Error conditions exist that the compiler can correct, but the program might not run correctly.
S	Severe error	Compilation continues, but object code is not generated. Error conditions exist that the compiler cannot correct.
U	Unrecoverable error	C The compiler halts. An internal compiler error has occurred. • 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 unrecoverable error. • If the unrecoverable error persists, report the message to your IBM service representative.

Related Concepts

"Compiler Message and Listing Information" on page 8

Related References

"Compiler Return Codes" on page 346
"Compiler Message Format" on page 346
"halt" on page 111
"maxerr" on page 173
"haltonmsg" on page 112

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.
- 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 halt compiler option has been detected.
40	An option error or an unrecoverable error has been detected.
41	A configuration file error has been detected.
250	An out-of-memory error has been detected. The **xlc++**command 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.

Related Concepts

"Compiler Message and Listing Information" on page 8

Related References

"Message Severity Levels and Compiler Response" on page 345
"Compiler Message Format"
"halt" on page 111
"maxerr" on page 173
"haltonmsg" on page 112

Compiler Message Format

Diagnostic messages have the following format when the **-qnosrcmsg** option is active (which is the default):

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

where:

file	is the name of the C or C++ source file with the error.
line_number	is the line number of the error.
column_number	is the column number for the error
15	is the compiler product identifier

dd is a two-digit code indicating the XL C/C++ component that issued the message. *dd* can have the following values:

- **00** - code generating or optimizing message
- **01** - compiler services message.
- **05** - message specific to the C compiler
- **06** - message specific to the C compiler
- **40** - message specific to the C++ compiler
- **86** - message specific to interprocedural analysis (IPA).

nnn is the message number

severity is a letter representing the severity of the error

text is a message describing the error

Diagnostic messages have the following format when the **-qsrcmsg** option is specified:

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

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

Related Concepts

"Compiler Message and Listing Information" on page 8

Related References

"Message Severity Levels and Compiler Response" on page 345
"Compiler Return Codes" on page 346
"halt" on page 111
"maxerr" on page 173
"haltonmsg" on page 112

Parallel Processing Support

This section contains information on environment variables and built-in functions used to control parallel processing. Topics in this section are:

- “OpenMP Run-time Options for Parallel Processing” on page 352
- “Built-in Functions Used for Parallel Processing” on page 354

Run-time Options for Parallel Processing

Run-time 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──=────option_and_args───┴──────────────────────────><
```

Parallelization run-time options can also be specified using OMP environment variables. When run-time options specified by OMP- and XLSMPOPTS-specific environment variables conflict, OMP options will prevail.

Note: You must use thread-safe compiler mode invocations when compiling parallelized program code.

Run-time option settings for the XLSMPOPTS environment variable are shown below, grouped by category:

Scheduling Algorithm Options

XLSMPOPTS Environment Variable Option	Description
schedule=*algorithm*=[*n*]	This option specifies the scheduling algorithm used for loops not explicitly assigned a scheduling algorithm. Valid options for *algorithm* are: • guided • affinity • dynamic • static If specified, the chunk size *n* must be an integer value of 1 or greater. The default scheduling algorithm is **static**.

Parallel Environment Options

XLSMPOPTS Environment Variable Option	Description
parthds=*num*	*num* represents the number of parallel threads 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*	*num* represents the number of user threads expected. This option should be used if the program code explicitly creates threads, in which case *num* should be set to the number of threads created. The default value for *num* is 0.
stack=*num*	*num* specifies the largest amount of space required for a thread's stack. The default value for *num* is 2097152. 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.

Performance Tuning Options

XLSMPOPTS Environment Variable Option	Description
spins=*num*	*num* represents 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.

XLSMPOPTS Environment Variable Option	Description
yields=*num*	*num* represents 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*	*num* represents 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

XLSMPOPTS Environment Variable Option	Description
profilefreq=*num*	*num* represents the sampling rate at which each loop is revisited to determine appropriateness for parallel processing. The run-time 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. 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. The maximum sampling rate is 32. Values of *num* exceeding 32 are changed to 32.
parthreshold=*mSec*	*mSec* specifies the expected running time in milliseconds below which a loop must be run sequentially. *mSec* can be specified using decimal places. If **parthreshold** is set to 0, a parallelized loop will never be serialized by the dynamic profiler. The default value for *mSec* is 0.2 milliseconds.
seqthreshold=*mSec*	*mSec* specifies the expected running time in milliseconds beyond which a loop that has been serialized by the dynamic profiler must revert to being run in parallel mode again. *mSec* can be specified using decimal places. The default value for *mSec* is 5 milliseconds.

Related Concepts

“Program Parallelization” on page 11
“OpenMP Directives” on page 11

Related References

"OpenMP Run-time Options for Parallel Processing"
"Built-in Functions Used for Parallel Processing" on page 354

For complete information about the OpenMP Specification, see:
OpenMP Web site at www.openmp.org
OpenMP Specification at www.openmp.org/specs

OpenMP Run-time Options for Parallel Processing

OpenMP run-time time 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.

Note: You must use thread-safe compiler mode invocations when compiling parallelized program code.

OpenMP run-time options fall into different categories as described below:

Scheduling Algorithm Environment Variable

OMP_SCHEDULE=*algorithm*	This option specifies the scheduling algorithm used for loops not explictly assigned a scheduling alogorithm with the **omp schedule** directive. For example: `OMP_SCHEDULE="guided, 4"` Valid options for *algorithm* are: • 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**.

Parallel Environment Environment Variables

OMP_NUM_THREADS=*num*	*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 during program execution by calling the **omp_set_num_threads( )** runtime library function. 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. 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	This environment variable enables or disables nested parallelism. The setting of this environment variable can be overrridden 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.

Dynamic Profiling Environment Variable

OMP_DYNAMIC=TRUE|FALSE	This 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 runtime to make the best use of system resources. If set to FALSE, dynamic adjustment is disabled. The default setting is TRUE.

Related Concepts

"Program Parallelization" on page 11
"OpenMP Directives" on page 11

Related References

"Pragmas to Control Parallel Processing" on page 323
"Built-in Functions Used for Parallel Processing"

For complete information about the OpenMP Specification, see:
- OpenMP Web site (www.openmp.org)
- OpenMP Specification (www.openmp.org/specs)

Built-in Functions Used for Parallel Processing

Use these built-in functions to obtain information about the parallel environment. Function definitions for the **omp_** functions can be found in the **omp.h** header file.

Function Prototype	Description
int omp_get_num_threads(void);	This function returns the number of threads currently in the team executing the parallel region from which it is called.
void omp_set_num_threads(int *num_threads*);	This function overrides the setting of the OMP_NUM_THREADS environment variable, and specifies the number of threads to use in parallel regions following this directive. The value num_threads must be a positive integer. If the num_threads clause is present, then for the parallel region it is applied to, it supersedes the number of threads requested by the omp_set_num_threads library function or the OMP_NUM_THREADS environment variable. Subsequent parallel regions are not affected by it.
int omp_get_max_threads(void);	This function returns the maximum value that can be returned by calls to omp_get_num_threads.
int omp_get_thread_num(void);	This function returns the thread number, within its team, of the thread executing the function. The thread number lies between 0 and omp_get_num_threads()-1, inclusive. The master thread of the team is thread 0.
int omp_get_num_procs(void);	This function returns the maximum number of processors that could be assigned to the program.
int omp_in_parallel(void);	This function returns non-zero if it is called within the dynamic extent of a parallel region executing in parallel; otherwise, it returns 0.
void omp_set_dynamic(int dynamic_threads);	This function enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.
int omp_get_dynamic(void);	This function returns non-zero if dynamic thread adjustments enabled and returns 0 otherwise.
void omp_set_nested(int nested);	This function enables or disables nested parallelism.
int omp_get_nested(void);	This function returns non-zero if nested parallelism is enabled and 0 if it is disabled.

Function Prototype	Description

void omp_init_lock(omp_lock_t **lock*);
 	
void omp_init_nest_lock(omp_nest_lock_t **lock*);
	These functions provide the only means of initializing a lock. Each function initializes the lock associated with the parameter *lock* for use in subsequent calls.

void omp_destroy_lock(omp_lock_t **lock*);
 	
void omp_destroy_nest_lock(omp_nest_lock_t **lock*);
	These functions ensure that the specified lock variable *lock* is uninitialized.

void omp_set_lock(omp_lock_t *lock);
 	
void omp_set_nest_lock(omp_nest_lock_t *lock);
	Each of these functions blocks the thread executing the function until the specified lock is available and then sets the lock. 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.

void omp_unset_lock(omp_lock_t *lock);
 	
void omp_unset_nest_lock(omp_nest_lock_t *lock);
	These functions provide the means of releasing ownership of a lock.

int omp_test_lock(omp_lock_t *lock);
 	
int omp_test_nest_lock(omp_nest_lock_t *lock);
	These functions attempt to set a lock but do not block execution of the thread.

double omp_get_wtime(void);	Returns the time elapsed from a fixed starting time. The value of the fixed starting time is determined at the start of the current program, and remains constant throughout program execution.
double omp_get_wtick(void);	Returns the number of seconds between clock ticks.

Note: 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.

For complete information about OpenMP runtime library functions, refer to the OpenMP C/C++ Application Program Interface specification.

Related Concepts

“Program Parallelization” on page 11

Related Tasks

“Control Parallel Processing with Pragmas” on page 31

Related References

“Pragmas to Control Parallel Processing” on page 323
“OpenMP Run-time Options for Parallel Processing” on page 352
Appendix B, “Built-in Functions,” on page 361

Part 4. Appendixes

Appendix A. Predefined Macros

Predefined macros fall into several categories: those related to language features, those related to the compiler, and those related to the Linux platform.

For information about macros related to language features, including those mandated by the language specification, see "Preprocessor Directives" in *XL C/C++ Language Reference*.

Macros indicating the XL compilers

Predefined macros related to the XL compiler are always defined.

Predefined macro name	Description
__IBMC__	C Indicates the level of the XL C compiler as an integer constant representing version, release, and modification number.
__IBMCPP__	C++ Indicates the level of the XL C++ compiler as an integer constant representing version, release, and modification number.
__xlc__	C Indicates the level of the XL C compiler as a string displaying the version, release, modification, and fix level.
__xlC__	C++ Indicates the level of the XL C++ compiler as a three-digit hexadecimal constant, representing version, release, and modification number. Using the XL C compiler also automatically defines this macro.

Macros related to the Linux platform

The following predefined macros are provided to facilitate porting applications between platforms.

Predefined Macro Name	Description
_ARCH_PPC	Defined to 1 if the processor architecture is PowerPC.
_ARCH_PPC64	Defined to 1 if the processor architecture is PowerPC.
__BASE_FILE__	Defined to the fully qualified filename of the primary source file.
_BIG_ENDIAN	Defined to 1.
__BIG_ENDIAN__	Defined to 1.
_CALL_SYSV	Defined to 1.
__CHAR_UNSIGNED__	Defined to 1 if the option **-qchars=unsigned** or **#pragma chars(unsigned)** is in effect. This macro is undefined if the option **-qchars=signed** or **#pragma chars(signed)** is in effect.
__ELF__	Defined to 1 on this platform to indicate the ELF object model is in effect.
__EXCEPTIONS	C++ Defined to 1 if the **-qeh** option is in effect. Otherwise it is not defined.
__GXX_WEAK__	Undefined for C. For C++, this macro is defined to 0 for g++ V3.3 or 1 for g++ V3.5.
__HOS_LINUX__	Defined to 1 if the host operating system is Linux. Otherwise it is not defined.
__linux	Defined to 1.
__linux__	Defined to 1.
__OPTIMIZE__	Defined to 2 for optimization level **-O** or **-O2**, or to 3 for optimization level **-O3** or higher.
__OPTIMIZE_SIZE__	Defined to 1 if the options **-qcompact** and **-O** are set. Otherwise it is not defined.
__powerpc	Defined to 1.

Predefined Macro Name	Description
__powerpc__	Defined to 1.
__powerpc64__	Defined to 1 when compiling in 64-bit mode. Otherwise it is not defined.
__PPC	Defined to 1.
__PPC__	Defined to 1.
__PPC64__	Defined to 1 when compiling in 64-bit mode. Otherwise it is not defined.
__SIZE_TYPE__	Defined to the underlying type of **size_t** on this platform. On this platform, the macro is defined as **long unsigned int**.In 32-bit mode, the macro is defined as **unsigned int**. In 64-bit mode, the macro is defined as **long int**. The compile mode is controlled by the **-q32** and **-q64** options.
__TOS_LINUX__	Defined to 1 if the target operating system is Linux. Otherwise it is not defined.
__unix	Defined to 1 on all UNIX-like platforms. Otherwise it is not defined.
__unix__	Defined to 1 on all UNIX-like platforms. Otherwise it is not defined.

Appendix B. Built-in Functions

The compiler provides you with a selection of built-in functions to help you write more efficient programs. This section summarizes the various built-in functions available to you.

You can also find additional built-in functions described at "Built-in Functions Used for Parallel Processing" on page 354.

Name	Prototype	Description
__alignx	void __alignx(int *alignment*, const void **address*);	Informs the compiler that the specified *address* is aligned at a known compile-time offset. *Alignment* must be a positive constant integer with a value greater than zero and of a power of two.
__bcopy	void __bcopy(char *, char *, int);	Block copy
__bzero	void __bzero(void *, size_t);	Block zero
__check_lock_mp	unsigned int __check_lock_mp (const int* *addr*, int *old_value*, int *new_value*);	Check Lock on MultiProcessor systems. Conditionally updates a single word variable atomically. *addr* specifies the address of the single word variable. *old_value* specifies the old value to be checked against the value of the single word variable. *new_value* specifies the new value to be conditionally assigned to the single word variable. The word variable must be aligned on a full word boundary. Return values: 1. A return value of false indicates that the single word variable was equal to the old value and has been set to the new value. 2. A return value of true indicates that the single word variable was not equal to the old value and has been left unchanged.

Name	Prototype	Description
__check_lockd_mp	unsigned int __check_lockd_mp (const long long int* *addr*, long long int *old_value*, long long int *new_value*);	Check Lock Doubleword on MultiProcessor systems. Conditionally updates a double word variable atomically. *addr* specifies the address of the double word variable. *old_value* specifies the old value to be checked against the value of the double word variable. *new_value* specifies the new value to be conditionally assigned to the double word variable. The double word variable must be aligned on a double word boundary. Return values: 1. A return value of false indicates that the double word variable was equal to the old value and has been set to the new value. 2. A return value of true indicates that the double word variable was not equal to the old value and has been left unchanged.
__check_lock_up	unsigned int __check_lock_up (const int* *addr*, int *old_value*, int *new_value*);	Check Lock on UniProcessor systems. Conditionally updates a single word variable atomically. *addr* specifies the address of the single word variable. *old_value* specifies the old value to be checked against the value of the single word variable. *new_value* specifies the new value to be conditionally assigned to the single word variable. The word variable must be aligned on a full word boundary. Return values: • A return value of false indicates that the single word variable was equal to the old value, and has been set to the new value. • A return value of true indicates that the single word variable was not equal to the old value and has been left unchanged.

Name	Prototype	Description
__check_lockd_up	unsigned int __check_lockd_up (const long long int* *addr*, long long int *old_value*, int long long *new_value*);	Check Lock Doubleword on UniProcessor systems. Conditionally updates a double word variable atomically. *addr* specifies the address of the double word variable. *old_value* specifies the old value to be checked against the value of the double word variable. *new_value* specifies the new value to be conditionally assigned to the double word variable. The double word variable must be aligned on a double word boundary. Return values: • A return value of false indicates that the double word variable was equal to the old value, and has been set to the new value. • A return value of true indicates that the double word variable was not equal to the old value and has been left unchanged.
__clear_lock_mp	void __clear_lock_mp (const int* *addr*, int *value*);	Clear Lock on MultiProcessor systems. Atomic store of the *value* into the single word variable at the address *addr*. The word variable must be aligned on a full word boundary.
__clear_lockd_mp	void __clear_lockd_mp (const long long int* *addr*, long long int *value*);	Clear Lock Doubleword on MultiProcessor systems. Atomic store of the *value* into the double word variable at the address *addr*. The double word variable must be aligned on a double word boundary.
__clear_lock_up	void __clear_lock_up (const int* *addr*, int *value*);	Clear Lock on UniProcessor systems. Atomic store of the *value* into the single word variable at the address *addr*. The word variable must be aligned on a full word boundary.
__clear_lockd_up	void __clear_lockd_up (const long long int* *addr*, long long int *value*);	Clear Lock Doubleword on UniProcessor systems. Atomic store of the *value* into the double word variable at the address *addr*. The double word variable must be aligned on a double word boundary.

Name	Prototype	Description
__cntlz4	unsigned int __cntlz4(unsigned int);	Count Leading Zeros, 4-Byte Integer
__cntlz8	unsigned int __cntlz8(unsigned long long);	Count Leading Zeros, 8-Byte Integer
__cnttz4	unsigned int __cnttz4(unsigned int);	Count Trailing Zeros, 4-Byte Integer
__cnttz8	unsigned int __cnttz8(unsigned long long);	Count Trailing Zeros, 8-Byte Integer
__dcbt( )	void __dcbt (void *);	Data Cache Block Touch. Loads the block of memory containing the specified address into the data cache.
__dcbz( )	void __dcbz (void *);	Data Cache Block set to Zero. Sets the specified address in the data cache to zero (0).
__eieio	void __eieio(void);	Extra name for the existing **__iospace_eieio** built-in. Compiler will recognize **__eieio** built-in. Everything except for the name is exactly same as for **__iospace_eieio**. **__eieio** is consistent with the corresponding PowerPC instruction name.
__exp	double __exp(double);	Returns the exponential value
__fabs	double __fabs(double);	Returns the absolute value
__fabss	float __fabss(float);	Returns the short floating point absolute value
__fcfid	double __fcfid (double);	Floating Convert From Integer Doubleword. The 64bit signed fixedpoint operand is converted to a double-precision floating-point.
__fctid	double __fctid (double);	Floating Convert to Integer Doubleword. The floating-point operand is converted into 64-bit signed fixed-point integer, using the rounding mode specified by $FPSCR_{RN}$ (Floating-Point Rounding Control field in the Floating-Point Status and Control Register).

Name	Prototype	Description
__fctidz	double __fctidz (double);	Floating Convert to Integer Doubleword with Rounding towards Zero. The floating-point operand is converted into 64-bit signed fixed-point integer, using the rounding mode Round toward Zero.
__fctiw	double __fctiw (double);	Floating Convert To Integer Word. The floating-point operand is converted to a 32-bit signed fixed-point integer, using the rounding mode specified by $FPSCR_{RN}$ (Floating-Point Rounding Control field in the Floating-Point Status and Control Register).
__fctiwz	double __fctiwz (double);	Floating Convert To Integer Word with Rounding towards Zero. The floating-point operand is converted to a 32-bit signed fixed-point integer, using the rounding mode Round toward Zero.
__fmadd	double __fmadd(double, double, double);	Floating point multiply-add
__fmadds	float __fmadds(float, float, float);	Floating point multiply-add short
__fmsub	double __fmsub(double, double, double);	Floating point multiply-subtract
__fmsubs	float __fmsubs(float, float, float);	Floating point multiply-subtract
__fnabs	double __fnabs(double);	Floating point negative absolute
__fnabss	float __fnabss(float);	Floating point negative absolute
__fnmadd	double __fnmadd(double, double, double);	Floating point negative multiply-add
__fnmadds	float __fnmadds (float, float, float);	Floating point negative multiply-add
__fnmsub	double __fnmsub(double, double, double);	Floating point negative multiply-subtract
__fnmsubs	float __fnmsubs (float, float, float);	__fnmsubs (*a*, *x*, *y*) = [- (*a* * *x* - *y*)]
__fre	double fre (double);	Floating point reciprocal __fre (*x*) = [(estimate of) 1.0/*x*] (POWER5 processors only)
__fres	float __fres (float);	Floating point reciprocal __fres (*x*) = [(estimate of) 1.0/*x*]
__frsqrte	double __frsqrte (double);	Floating point reciprocal square root __frsqrte (*x*) = [(estimate of) 1.0/sqrt(*x*)]

Name	Prototype	Description
__frsqrtes	float __frsqrtes (float);	Floating point reciprocal square root __frsqrtes (*x*) = [(estimate of) 1.0/sqrt(*x*)] (POWER5 processors only)
__fsel	double __fsel (double, double, double);	Floating point select if (*a* >= 0.0) then __fsel (*a*, *x*, *y*) = *x*; else __fsel (*a*, *x*, *y*) = *y*
__fsels	float __fsels (float, float, float);	Floating point select if (*a* >= 0.0) then __fsels (*a*, *x*, *y*) = *x*; else __fsels (*a*, *x*, *y*) = *y*
__fsqrt	double __fsqrt (double);	Floating point square root __fsqrt (*x*) = square root of *x*
__fsqrts	float __fsqrts (float);	Floating point square root __fsqrts (*x*) = square root of *x*
__iospace_eieio	void __iospace_eieio(void);	Generates an EIEIO instruction
__iospace_lwsync	(equivalent to: void __iospace_lwsync(void);)	Generates a lwsync instruction
__iospace_sync	(equivalent to: void __iospace_sync(void);)	Generates a sync instruction
__isync	void __isync(void);	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.
__load2r	unsigned short __load2r(unsigned short*);	Load halfword byte reversed
__load4r	unsigned int __load4r(unsigned int*);	Load word byte reversed
__lwsync	(Compiler will recognize **__lwsync** built-in.)	Extra name for the existing **__iospace_lwsync** built-in. Compiler will recognize **__lwsync** built-in. Everything except for the name is exactly same as for **__iospace_lwsync**. **__lwsync** is consistent with the corresponding PowerPC instruction name. This function is supported only by the PowerPC 970 processor. Value must be known at compile time.
__mfdcr	unsigned __mfdcr(const int);	Returns value of device control register. Valid only for PowerPC 440.
__mtdcr	void __mtdcr(const int, unsigned long);	Set value of device control register with unsigned long value. Value must be known at compile time. Valid only for PowerPC 440.

Name	Prototype	Description
__mfspr	unsigned __mfspr(const int);	Returns value of special purpose register. Value of "const int" must be known at compile time.
__mtspr	void __mtspr(const int, unsigned long);	Set value of special purpose register with unsigned long value. Values must be known at compile time.
__mtfsb0	void __mtfsb0(unsigned int *bt*);	Move to FPSCR Bit 0. Bit *bt* of the FPSCR is set to 0. *bt* must be a constant and 0<=*bt*<=31.
__mtfsb1	void __mtfsb1(unsigned int *bt*);	Move to FPSCR Bit 1. Bit *bt* of the FPSCR is set to 1. *bt* must be a constant and 0<=*bt*<=31.
__mtfsf	void __mtfsf(unsigned int *flm*, unsigned int *frb*);	Move to FPSCR Fields. The contents of *frb* are placed into the FPSCR under control of the field mask specified by *flm*. The field mask *flm* identifies the 4bit fields of the FPSCR affected. *flm* must be a constant 8–bit mask.
__mtfsfi	void __mtfsfi(unsigned int *bf*, unsigned int *u*);	Move to FPSCR Field Immediate. The value of the *u* is placed into FPSCR field specified by *bf*. *bf* and *u* must be constants, with 0<=*bf*<=7 and 0<=*u*<=15.
__mulhd	long long int __mulhd(long long int *ra*, long long int *rb*);	Multiply High Doubleword Signed. Returns the highorder 64 bits of the 128bit product of the operands *ra* and *rb*.
__mulhdu	unsigned long long int __mulhdu(unsigned long long int *ra*, unsigned long long int *rb*);	Multiply High Doubleword Unsigned. Returns the highorder 64 bits of the 128bit product of the operands *ra* and *rb*.
__mulhw	int __mulhw(int *ra*, int *rb*);	Multiply High Word Signed. Returns the highorder 32 bits of the 64bit product of the operands *ra* and *rb*.
__mulhwu	unsigned int __mulhwu(unsigned int *ra*, unsigned int *rb*);	Multiply High Word Unsigned. Returns the highorder 32 bits of the 64bit product of the operands *ra* and *rb*.

Name	Prototype	Description
__parthds	int __parthds(void);	Returns the value of the parthds run-time option. If the parthds option is not explicitly set by the user, the function returns the default value set by the run-time library. If the -qsmp compiler option was not specified during program compilation, this function returns 1 regardless of run-time options selected.
__popcnt4	int __popcnt4(unsigned int);	Returns the number of bits set for a 32-bit integer
__popcnt8	int __popcnt8(unsigned long long);	Returns the number of bits set for a 64-bit integer
__poppar4	int __poppar4(unsigned int);	Returns 1 if there is an odd number of bits set in a 32-bit integer. Returns 0 otherwise.
__poppar8	int __poppar8(unsigned long long);	Returns 1 if there is an odd number of bits set in a 64-bit integer. Returns 0 otherwise.
__pow	double __pow(double, double);	
__prefetch_by_load	void __prefetch_by_load(const void*);	Touch a memory location via explicit load
__prefetch_by_stream	void __prefetch_by_stream(const int, const void*);	Touch a memory location via explicit stream
__protected_stream_count	void __protected_stream_count(unsigned int *unit_cnt*, unsigned int *ID*);	Sets *unit_cnt* number of cache lines for the limited length protected stream with identifier *ID*. *Unit_cnt* must be an integer with value of 0 to 1023. Stream *ID* must have integer value 0 to 15. (POWER5 processors only)
__protected_stream_go	void __protected_stream_go();	Starts prefetching all limited-length protected streams. (POWER5 processors only)

Name	Prototype	Description
__protected_stream_set	void __protected_stream_set(unsigned int *direction*, const void* *addr*, unsigned int *ID*);	Establishes a limited length protected stream using identifier *ID*, which begins with the cache line at *addr* and then depending on the value of *direction*, fetches from either incremental (forward) or decremental (backward) memory addresses. The stream is protected from being replaced by any hardware detected streams. *Direction* must have value of 1 (forward) or 3 (backward). Stream *ID* must have integer value 0 to 15. (POWER5 processors only)
__protected_unlimited_stream_set_go	void __protected_unlimited_stream_set_go (unsigned int *direction*, const void* *addr*, unsigned int *ID*);	Establishes an unlimited length protected stream using identifier *ID*, which begins with the cache line at *addr* and then depending on the value of *direction*, fetches from either incremental (forward) or decremental (backward) memory addresses. The stream is protected from being replaced by any hardware detected streams. *Direction* must have value of 1 (forward) or 3 (backward). Stream *ID* must have integer value 0 to 15. (PowerPC 970 and POWER5 processors only)
__protected_stream_stop	void __protected_stream_stop(unsigned int *ID*);	Stops prefetching the protected steam with identifier *ID*. (POWER5 processors only)
__protected_stream_stop_all	void __protected_stream_stop_all();	Stops prefetching all protected steams. (POWER5 processors only)
__rdlam	unsigned long long __rdlam(unsigned long long *rs*, unsigned int *shift*, unsigned long long *mask*);	Rotate Double Left and AND with Mask. The contents of *rs* are rotated left *shift* bits. The rotated data is ANDed with the mask and returned as a result. *mask* must be a constant and represent a contiguous bit field.
__readflm	double __readflm();	Read floating point status/control register

Name	Prototype	Description
__rldimi	unsigned long long __rldimi(unsigned long long *rs*, unsigned long long *is*, unsigned int *shift*, unsigned long long *mask*);	Rotate Left Doubleword Immediate then Mask Insert. Rotates *rs* left *shift* bits then inserts *rs* into *is* under bit mask *mask*. Shift must be a constant and 0<=*shift*<=63. *mask* must be a constant and represent a contiguous bit field.
__rlwimi	unsigned int __rlwimi(unsigned int *rs*, unsigned int *is*, unsigned int *shift*, unsigned int *mask*);	Rotate Left Word Immidiate then Mask Insert. Rotates *rs* left *shift* bits then inserts *rs* into *is* under bit mask *mask*. Shift must be a constant and 0<=*shift*<=31. *mask* must be a constant and represent a contiguous bit field.
__rlwnm	unsigned int __rlwnm(unsigned int *rs*, unsigned int *shift*, unsigned int *mask*);	Rotate Left Word then AND with Mask. Rotates *rs* left *shift* bits, then ANDs *rs* with bit mask *mask*. *mask* must be a constant and represent a contiguous bit field.
__rotatel4	unsigned int __rotatel4(unsigned int *rs*, unsigned int *shift*);	Rotate Left Word. Rotates *rs* left *shift* bits.
__setflm	double __setflm(double);	Set Floating Point Status/Control Register
__setrnd	double __setrnd(int);	Set Rounding Mode
__stfiw	void __stfiw(const int* *addr*, double *value*);	Store Floating-Point as Integer Word. The contents of the loworder 32 bits of *value* are stored, without conversion, into the word in storage addressed by *addr*.
__store2r	void __store2r(unsigned short, unsigned short *);	Store 2-byte register
__store4r	void __store4r(unsigned int, unsigned int *);	Store 4-byte register

Name	Prototype	Description
__swdiv_nochk	double __swdiv_nochk(double, double);	Floating-point division of **double** types; no range checking. 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. Argument restrictions: numerators equal to infinity are not allowed; denominators equal to infinity, zero, or denormalized are not allowed; the quotient of numerator and denominator may not be equal to positive or negative infinity. With **-qstrict** in effect, the result is identical bitwise to IEEE division. For correct operation in this scenario, the arguments must satisfy the following additional restrictions. Numerators must have an absolute value greater than 2 ^ (-970) and less than infinity. Denominators must have an absolute value greater than 2 ^ (-1022) and less than 2 ^ 1021. The quotient of numerator and denominator must have an absolute value greater than 2 ^ (-1021) and less than 2 ^ 1023.
__swdiv	double __swdiv(double, double);	Floating-point division of **double** types. No argument restrictions.
__swdivs_nochk	float __swdivs_nochk(float, float);	Floating-point division of **float** types; no range checking. Argument restrictions: numerators equal to infinity are not allowed; denominators equal to infinity, zero, or denormalized are not allowed; the quotient of numerator and denominator may not be equal to positive or negative infinity.
__swdivs	float __swdivs(float, float);	Floating-point division of **double** types. No argument restrictions.
__sync	void __sync(void);	Extra name for the existing **__iospace_sync** built-in. Compiler will recognize **__sync** built-in. Everything except for the name is exactly same as for **__iospace_sync**. **__sync** is consistent with the corresponding PowerPC instruction name.

Name	Prototype	Description
__tdw	void __tdw(long long *a*, long long *b*, unsigned int *TO*);	Trap Doubleword. Operand *a* is compared with operand *b*. This comparison results in five conditions which are ANDed with a 5-bit constant *TO* containing a value of **0** to **31** inclusive. If the result is not 0 the system trap handler is invoked. Each bit positon, if set, indicates one or more of the following possible conditions: **0 (high-order bit)** *a* Less Than *b*, using signed comparison. **1** *a* Greater Than *b*, using signed comparison. **2** *a* Equal *b* **3** *a* Less Than *b*, using unsigned comparison. **4 (low order bit)** *a* Greater Than *b*, using unsigned comparison.
__trap	void __trap(int);	Trap
__trapd	void __trapd (longlong);	Trap if the parameter is not zero.
__tw	void __tw(int *a*, int *b*, unsigned int *TO*);	Trap Word. Operand ***a*** is compared with operand ***b***. This comparison results in five conditions which are ANDed with a 5-bit constant *TO* containing a value of **0** to **31** inclusive. If the result is not 0 the system trap handler is invoked. Each bit positon, if set, indicates one or more of the following possible conditions: **0 (high-order bit)** *a* Less Than *b*, using signed comparison. **1** *a* Greater Than *b*, using signed comparison. **2** *a* Equal *b* **3** *a* Less Than *b*, using unsigned comparison. **4 (low order bit)** *a* Greater Than *b*, using unsigned comparison.

Name	Prototype	Description
__usrthds	int __usrthds(void);	Returns the value of the **usrthds** run-time option. If the **usrthds** option is not explicitly set by the user, or the -qsmp compiler option was not specified during program compilation, this function returns 0 regardless of run-time options selected.

Appendix C. Libraries in XL C/C++

Redistributable libraries

XL C/C++ provides the following redistributable libraries. Depending on your application, you may need to ship one or more of these libraries together with applications built with XL C/C++.

libibmc++.so
Used only by C++ programs.

libxlsmp.so libxlsmp_ser.so libxlsmpdebug.so
Required when **-qsmp** or **-qsmp=omp** options are in effect.

Order of linking

XL C/C++ links libraries in the following order:

1. user .o files and libraries
2. XL C/C++ libraries
3. C++ standard libraries
4. C standard libraries

The table below shows the linking order in greater detail for a ″Hello World″ type of program.

Directory paths shown may vary depending on your particular compiler configuration. See the default configuration file installed on your system for information specific to your particular compiler configuration. See "Specify Compiler Options in a Configuration File" on page 24 for more information about compiler default configuration files in general.

ld Command Components	Options	ld Arguments	xldriver attributes
ld	gcc, g++	collect2	ld / ld_64
	xlc, xlC	ld	
enable exception handling personality handlers	all	--eh-frame-hdr	Option added to command line by xldriver
generate .ident directives	-Qn		
	otherwise	-Qy	Option added to command line by xldriver
output kind	-shared -static	-shared	Option added to command line by xldriver
	-shared	-shared	
	-static	-static	
	Otherwise		
arch	32-bit	-melf32ppclinux	Option added to command line by xldriver
	64-bit	-mel64ppc	
dynamic loader	32-bit !-shared !-static	-dynamic-linker /lib/ld.so.1	dynlib
	64-bit !-shared !-static	-dynamic-linker /lib64/ld64.so.1	dynlib64

ld Command Components	Options	ld Arguments	xldriver attributes
call to main( )	32-bit !-shared	/usr/libcrt1.o	crt
	64-bit !-shared	/opt/cross/powerpc64-linux/lib/crt1.o	crt_64
	32-bit !-shared -p	/usr/lib/gcrt1.o	mcrt
	32-bit !-shared -pg		gcrt
	64-bit !-shared -p	/opt/cross/powerpc64-linux/lib/gcrt1.o	mcrt_64
	64-bit !-shared -pg		gcrt_64
init/fini functions prolog	32-bit all	/usr/lib/crti.o	crtp
	64-bit all	/opt/cross/powerpc64-linux/lib/crti.o	crtp_64
init/fini register	-shared -static	crtbeginT.o	crtbegin_t / crtbegin_t_64
	-static		
	-shared	crtbeginS.o	crtbegin_s / crtbegin_s_64
	otherwise	crtbegin.o	crtbegin / crtbegin_64
library search paths	32-bit gcc	-L<*gcc*>/gcc-lib	gcc_libdirs
	64-bit gcc	-L<*gcc64*>/gcc-lib	gcc_libdirs_64
	32-bit g++	-L<*gcc*>/gcc-lib/powerpc-suse-linux-gnu/3.2 -L<*gcc*>/gcc-lib	gcc_libdirs
	64-bit g++	-L*gcc64*/gcc-lib/powerpc64-linux-gnu/3.2 -L*gcc64*/gcc-lib	gcc_libdirs_64
user .o files and libraries	all		
vacpp libraries	all		libraries2 / libraries2_64
C++ standard libraries	g++	-lstdc++ -lm	gcc_cpp_libs / gcc_cpp_libs_64
C standard libraries	gcc -static -static -shared-libgcc -shared -static-libgcc	-lgcc -lgcc_eh -lc -lgcc -lgcc_eh	gcc_static_libs / gcc_static_libs_64
	g++ -shared-libgcc	-lgcc_s -lgcc -lc -lgcc_s -lgcc	gcc_shared_libs / gcc_shared_libs_64
	all		gcc_libs / gcc_libs_64
save/restore routines	all	crtsavres.o	crtsavres / crtsavres_64
init/fini run		crtend.o	crtend / crtend_64
	-shared	crtendS.o	crtend_s / crtend_s_64
init/fini functions epilog	all	/usr/lib/crtn.o	crte
		/opt/cross/powerpc64-linux/lib/crtn.o	crte_64

Appendix D. Problem Solving

Topics in this section are:

- “Message Catalog Errors”
- “Correcting Paging Space Errors During Compilation” on page 378

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 IBM XL C/C++ internal message number. This message is issued in English only.

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

```
ls /usr/lib/nls/msg/%L/*.cat
```

where **%L** is the current primary language environment (locale) setting. The default locale is **C**. The locale for United States English is **en_US**.

The default message catalogs in **/opt/ibmcmp/vacpp/7.0/msg** are called when:

- The compiler cannot find message catalogs for the locale specified by **%L**.
- 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.

Related Tasks

“Set Environment Variables” on page 17
“Set Other Environment Variables” on page 18

Correcting Paging Space Errors During Compilation

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

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

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

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

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

Appendix E. ASCII Character Set

XL C/C++ uses the American National Standard Code for Information Interchange (ASCII) character set.

The following table lists the standard ASCII characters in ascending numerical order, with their corresponding decimal, octal, and hexadecimal values. It also shows the control characters with **Ctrl-** notation. For example, the carriage return (ASCII symbol **CR**) appears as **Ctrl-M**, which you enter by simultaneously pressing the **Ctrl** key and the **M** key.

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Meaning
0	0	00	Ctrl-@	NUL	null
1	1	01	Ctrl-A	SOH	start of heading
2	2	02	Ctrl-B	STX	start of text
3	3	03	Ctrl-C	ETX	end of text
4	4	04	Ctrl-D	EOT	end of transmission
5	5	05	Ctrl-E	ENQ	enquiry
6	6	06	Ctrl-F	ACK	acknowledge
7	7	07	Ctrl-G	BEL	bell
8	10	08	Ctrl-H	BS	backspace
9	11	09	Ctrl-I	HT	horizontal tab
10	12	0A	Ctrl-J	LF	new line
11	13	0B	Ctrl-K	VT	vertical tab
12	14	OC	Ctrl-L	FF	form feed
13	15	0D	Ctrl-M	CR	carriage return
14	16	0E	Ctrl-N	SO	shift out
15	17	0F	Ctrl-O	SI	shift in
16	20	10	Ctrl-P	DLE	data link escape
17	21	11	Ctrl-Q	DC1	device control 1
18	22	12	Ctrl-R	DC2	device control 2
19	23	13	Ctrl-S	DC3	device control 3
20	24	14	Ctrl-T	DC4	device control 4
21	25	15	Ctrl-U	NAK	negative acknowledge
22	26	16	Ctrl-V	SYN	synchronous idle
23	27	17	Ctrl-W	ETB	end of transmission block
24	30	18	Ctrl-X	CAN	cancel
25	31	19	Ctrl-Y	EM	end of medium
26	32	1A	Ctrl-Z	SUB	substitute
27	33	1B	Ctrl-[	ESC	escape
28	34	1C	Ctrl-\	FS	file separator

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Meaning
29	35	1D	Ctrl-]	GS	group separator
30	36	1E	Ctrl-^	RS	record separator
31	37	1F	Ctrl-_	US	unit separator
32	40	20		SP	digit select
33	41	21		!	exclamation point
34	42	22		"	double quotation mark
35	43	23		#	pound sign, number sign
36	44	24		$	dollar sign
37	45	25		%	percent sign
38	46	26		&	ampersand
39	47	27		'	apostrophe
40	50	28		(	left parenthesis
41	51	29		)	right parenthesis
42	52	2A		*	asterisk
43	53	2B		+	addition sign
44	54	2C		,	comma
45	55	2D		-	subtraction sign
46	56	2E		.	period
47	57	2F		/	right slash
48	60	30		0	
49	61	31		1	
50	62	32		2	
51	63	33		3	
52	64	34		4	
53	65	35		5	
54	66	36		6	
55	67	37		7	
56	70	38		8	
57	71	39		9	
58	72	3A		:	colon
59	73	3B		;	semicolon
60	74	3C		<	less than
61	75	3D		=	equal
62	76	3E		>	greater than
63	77	3F		?	question mark
64	100	40		@	at sign
65	101	41		A	
66	102	42		B	
67	103	43		C	

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Meaning
68	104	44		D	
69	105	45		E	
70	106	46		F	
71	107	47		G	
72	110	48		H	
73	111	49		I	
74	112	4A		J	
75	113	4B		K	
76	114	4C		L	
77	115	4D		M	
78	116	4E		N	
79	117	4F		O	
80	120	50		P	
81	121	51		Q	
82	122	52		R	
83	123	53		S	
84	124	54		T	
85	125	55		U	
86	126	56		V	
87	127	57		W	
88	130	58		X	
89	131	59		Y	
90	132	5A		Z	
91	133	5B		[	left bracket
92	134	5C		\	left slash, backslash
93	135	5D		]	right bracket
94	136	5E		^	hat, circumflex, caret
95	137	5F		_	underscore
96	140	60		`	grave accent
97	141	61		a	
98	142	62		b	
99	143	63		c	
100	144	64		d	
101	145	65		e	
102	146	66		f	
103	147	67		g	
104	150	68		h	
105	151	69		i	
106	152	6A		j	
107	153	6B		k	

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Meaning
108	154	6C		l	
109	155	6D		m	
110	156	6E		n	
111	157	6F		o	
112	160	70		p	
113	161	71		q	
114	162	72		r	
115	163	73		s	
116	164	74		t	
117	165	75		u	
118	166	76		v	
119	167	77		w	
120	170	78		x	
121	171	79		y	
122	172	7A		z	
123	173	7B		{	left brace
124	174	7C		|	logical or, vertical bar
125	175	7D		}	right brace
126	176	7E		~	similar, tilde
127	177	7F		DEL	delete

Notices

Note to U.S. Government Users Restricted Rights -- use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

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

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

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

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

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

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

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

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

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

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

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

This information is for planning purposes only. The information herein is subject to change before the products described become available.

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

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples 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:

Programming Interface Information

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

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

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

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

Trademarks and Service Marks

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

AIX
IBM
PowerPC
pSeries
SAA
VisualAge

Other company, product, and service names, which may be denoted by a double asterisk(**), may be trademarks or service marks of others.

Industry Standards

The following standards are supported:

- The C language is consistent with the International Standard for Information Systems-Programming Language C (ISO/IEC 9899-1999 (E)).
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).
- The C and C++ languages are consistent with the OpenMP C and C++ Application Programming Interface, Version 1.0.

SC09-7942-01