

IBM XL C/C++ Advanced Edition for Mac OS X



Compiler Reference

Version 6.0

IBM XL C/C++ Advanced Edition for Mac OS X



Compiler Reference

Version 6.0

Before using this information and the product it supports, be sure to read the information in "Notices" on page 319.

December 2003 - First Edition

This edition applies to Version 6 Release 0 of IBM XL C/C++ Advanced Edition for Mac OS X (Program Number 5724-G12) 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.

© Copyright International Business Machines Corporation 1995, 2003. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

How to read syntax diagrams	vii
Symbols	vii
Syntax items	vii
Syntax examples	viii

Part 1. Overview 1

XL C/C++ Advanced Edition for Mac OS

X Compiler	3
Compiler Modes	3
Compiler Options.	4
Types of Input Files	6
Types of Output Files	7
Compiler Message and Listing Information	8
Compiler Messages	8
Compiler Listings.	8

Part 2. Configuration and Use 9

Set Up the Compilation Environment 11

Set Environment Variables	11
Set Environment Variables in bash.	11
Set Other Environment Variables	11

Invoke the Compiler 13

Invoke the Linkage Editor	13
-------------------------------------	----

Specify Compiler Options 15

Specify Compiler Options on the Command Line.	15
-q Options.	15
Flag Options	16
Specify Compiler Options in Your Program Source Files	17
Specify Compiler Options in a Configuration File.	17
Tailor a Configuration File	18
Configuration File Attributes	18
Specify Compiler Options for Architecture-Specific Compilation	19
Resolving Conflicting Compiler Options.	19

Specify Path Names for Include Files 21

Directory Search Sequence for Include Files Using Relative Path Names	21
---	----

Structure a Program that Uses

Templates	23
Declaration of Stack in stack.h	23
Declaration of operator Functions in stack.c	24
Template Functions Declared Inline and Template Functions With Internal Linkage	24
Template Functions Defined within the Compilation Unit	25

Use the Template Registry to Define Template Functions	26
Recompiling Parts of Your Program After Making Source Changes	26
Define Template Functions Directly in Compilation Units	27
Generate Template Functions Automatically	28
How the Compiler Generates the Function Definitions.	29
Specifying the tempinc file	29
Specifying a Different Path for the tempinc Subdirectory	30
Regenerating the Template Instantiation File	30
Breaking a Template Instantiation File into More Than One File	30
Contents of Template Instantiation File	30
Using #pragma Directives in Header Files	31
Considerations for Shared Libraries	31

Part 3. Reference 33

Compiler Options 35

Compiler Command Line Options.	35
+ (plus sign)	44
# (pound sign)	45
aggrcopy	46
alias	47
align.	49
alloca	53
altivec	54
ansialias	55
arch	56
assert	57
attr	58
B.	59
bitfields	60
bundle	61
bundle_loader	62
C.	63
c	64
c_stdinc	65
cache	66
chars	68
check	69
cinc	71
common	72
compact	74
complexgccincl	75
plusplus	76
cpp_stdinc.	80
crt	81
D.	82
dbxextra	83
digraph.	84
dollar	86

E	87	rtti	195
eh	89	rwvftable	196
enum	90	s	197
F	93	showinc	198
flag	94	smallstack	199
float	95	source	200
flttrap	98	sourcetype	201
framework	100	spill	202
frameworkdir	101	srcmsg	203
fullpath	102	staticinline	204
g	103	statsym	205
gcc_c_stdinc	104	stdframework	206
gcc_cpp_stdinc	105	stdinc	207
genproto	106	strict	208
halt	107	strict_induction	209
haltonmsg	108	suppress	210
hot	109	symtab	211
I	110	syntaxonly	212
idirfirst	111	t	213
ignerrno	112	tabsize	214
ignprag	113	tempinc	215
info	114	templaterecompile	216
initauto	118	templateregistry	217
inline	119	tempmax	218
ipa	123	threaded	219
isolated_call	130	tmplparse	220
keyword	131	trigraph	221
L	132	tune	223
l	133	U	224
langlvl	134	unroll	225
lib	150	unwind	227
libansi	151	upconv	228
linedebug	152	V	229
list	153	v	230
listopt	154	vftable	231
longlong	155	vrsave	232
M	156	W	233
ma	158	w	234
macpstr	159	warnfourcharconsts	235
makedep	161	xcall	236
maxerr	163	xref	237
maxmem	165	y	238
mbsc, dbcs	166	Z	239
mkshrobj	167	General Purpose Pragmas	240
O, optimize	171	#pragma align	242
o	175	#pragma alloca	243
P	176	#pragma altivec_vrsave	244
p	177	#pragma chars	245
path	178	#pragma comment	246
pdf1, pdf2	179	#pragma complexgcc	247
pg	182	#pragma define	248
phsinfo	183	#pragma disjoint	249
pic	184	#pragma enum	250
print	185	#pragma execution_frequency	254
priority	186	#pragma hashome	256
proto	187	#pragma ibm_snapshot	257
Q	188	#pragma implementation	258
r	191	#pragma info	259
report	192	#pragma ishome	262
ro	193	#pragma isolated_call	263
roconst	194	#pragma langlvl	265

#pragma leaves	267
#pragma map	268
#pragma mc_func	270
#pragma options	272
#pragma option_override	278
#pragma pack	279
#pragma priority	281
#pragma reachable	282
#pragma reg_killed_by	283
#pragma report	285
#pragma strings	287
#pragma unroll	288
Acceptable Compiler Mode and Processor Architecture Combinations	290
Compiler Messages	293
Message Severity Levels and Compiler Response	293
Compiler Return Codes	294
Compiler Message Format	294

Part 4. Appendixes 297

Appendix A. Predefined Macros 299

Macros related to the platform	299
--	-----

Appendix B. Built-in Functions 301

**Appendix C. Libraries in XL C/C++
Advanced Edition for Mac OS X 311**

Redistributable Libraries	311
Order of Linking	311

Appendix D. Problem Solving 313

Message Catalog Errors	313
Correcting Paging Space Errors During Compilation	314

Appendix E. ASCII Character Set 315

Notices 319

Programming Interface Information	321
Trademarks and Service Marks	321
Industry Standards	321

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

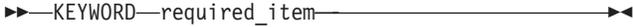
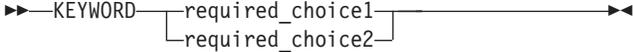
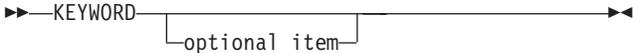
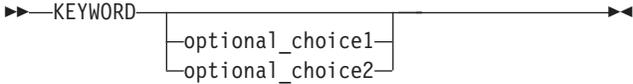
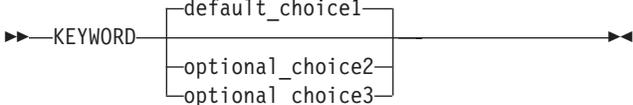
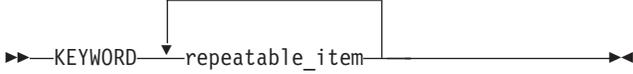
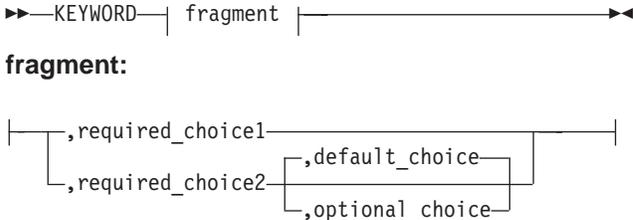
Item	Syntax example
Required item.	
Required items appear on the main path of the horizontal line. You must specify these items.	
Required choice.	
A required choice (two or more items) appears in a vertical stack on the main path of the horizontal line. You must choose one of the items in the stack.	
Optional item.	
Optional items appear below the main path of the horizontal line.	
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.	
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.	
Variable.	
Variables appear in lowercase italics. They represent names or values.	

Table 1. Syntax examples (continued)

Item	Syntax example
<p>Repeatable item.</p> <p>An arrow returning to the left above the main path of the horizontal line indicates an item that can be repeated.</p> <p>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.</p>	
<p>Fragment.</p> <p>The <code>— fragment —</code> symbol indicates that a labelled group is described below the main syntax diagram. Syntax is occasionally broken into fragments if the inclusion of the fragment would overly complicate the main syntax diagram.</p>	

Part 1. Overview

XL C/C++ Advanced Edition for Mac OS X Compiler

You can use IBM XL C/C++ Advanced Edition for Mac OS X 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++ Advanced Edition for Mac OS X compiler, see the following topics in this section:

- “Compiler Modes”
- “Compiler Options” on page 4
- “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++ Advanced Edition for Mac OS X 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
<code>xlc++</code>	<code>xlc++_r</code>
<code>xlC</code>	<code>xlC_r</code>
<code>xlc</code>	<code>xlc_r</code>
<code>cc</code>	<code>cc_r</code>
<code>c99</code>	<code>c99_r</code>
<code>c89</code>	<code>c89_r</code>
<code>gxlc++</code>	
<code>gxlc</code>	

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

xlc++ xlc	Invokes the compiler so that source files are compiled as C++ language source code. Files with <code>.c</code> suffixes, assuming you have not used the <code>-+</code> compiler option, are compiled as C language source code with a default language level of extc89 . If any of your source files are C++, you must use this invocation to link with the correct runtime libraries.
xlc	Invokes the compiler for C source files with a default language level of extc89 , and with options -qansialias , -qcplusplus , -qkeyword=inline , and -qnotrigraph set.
cc	Invokes the compiler for C source files with a default language level of extended and compiler options -qcplusplus , -qkeyword=inline , -qnorono , -qnoronoconst , and -qnotrigraph . Use this invocation for legacy C code that does not require compliance with ANSI C.
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. Note: The compiler supports all language features specified in the ISO C99 Standard. Note that the Standard also specifies features in the run-time library. These features may not be supported in the current run-time library and operating environment.
c89	Invokes the compiler for C source files, with a default language level of C89 . This invocation also sets compiler options -qnonlonglong , -qansialias (to allow type based aliasing) and -D_ANSI_SOURCE -D__STRICT_ANSI__ (for ANSI-conformant headers). Use this invocation for strict conformance to the ANSI standard (ISO/IEC 9899:1990).
gxlc++	You can use this utility to compile C++ files. It accepts many common gcc command line options, maps them to their xlc++ option equivalents, and then invokes xlc . For more information, refer to the <i>Reusing GNU C and C++ compiler options with gxlc and gxlc++</i> section in <i>Getting Started with XL C/C++</i> .
gxlc	You can use this utility to compile C files. It accepts many common gcc command line options, maps them to their xlc option equivalents, and then invokes xlc . For more information, refer to the <i>Reusing GNU C and C++ compiler options with gxlc and gxlc++</i> section in <i>Getting Started with XL C/C++</i> .
_r-suffixed Invocations	All _r -suffixed invocations additionally set the macro name -D_REENTRANT 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 13

Related References

“Compiler Command Line Options” on page 35

“General Purpose Pragmas” on page 240

“threaded” on page 219

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++ Advanced Edition for Mac OS X 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 19.

Related Tasks

“Invoke the Compiler” on page 13

“Specify Compiler Options” on page 15

“Resolving Conflicting Compiler Options” on page 19

Related References

“Compiler Command Line Options” on page 35

“General Purpose Pragmas” on page 240

Types of Input Files

You can input the following types of files to the XL C/C++ Advanced Edition for Mac OS X compiler:

C and C++ Source Files

These are files containing a C or C++ source module.

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.

The file system used by Mac OS X is partially case-insensitive. It preserves the case used to create the file, but does not distinguish different files by case. The file names `Hello.c` and `Hello.C` in the same directory refer to the same physical file. The compiler will recognize the `.c` versus `.C` filename extensions. However, if your source filenames do not conform to this file naming convention, you will need to use the `-qsourcetype` compiler option.

The `-qsourcetype` option instructs the compiler to not rely on the case in the filename suffix, and to instead assume a source type as specified by the option.

The compiler will also accept source files with the `.i` suffix. This extension designates preprocessed source 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 `xlcpp`. 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 `xlcpp` command preprocesses and compiles all the specified source files. Although you will usually want to use this default, you can use the `xlcpp` command 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.

Preprocessed Source Files

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

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

Object Files

Object files must have a `.o` suffix, for example, `year.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 <code>.s</code> suffix, for example, <code>check.s</code> . Assembler files are assembled to create an object file.
Shared Library Files	Shared library files must have a <code>.dylib</code> suffix, for example, <code>my_shrllib.dylib</code> .

Related Concepts

“Types of Output Files”

Types of Output Files

You can specify the following types of output files when invoking the XL C/C++ Advanced Edition for Mac OS X compiler.

Executable File By default, executable files are named `a.out`. To name the executable file something else, use the `-ofile_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 Object files must have a `.o` suffix, for example, `year.o`, unless the `-ofilename` option is specified.

If you specify the `-c` option, an output object file, `file_name.o`, is produced for each input source file `file_name.c`. 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 using the `xlcpp` command.

Assembler Files Assembler files must have a `.s` suffix, for example, `check.s`.

Assembler files are assembled to create an object file.

Preprocessed Source Files Preprocessed source files have a `.i` suffix, for example, `tax_calc.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, `file_name.c`.

Listing Files Listing files have a `.lst` suffix, for example, `form.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 File Shared library files have a `.dylib` suffix, for example, `my_shrllib.dylib`.

Target File 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

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.

The compiler also places messages in the source listing if you specify the **-qsource** option.

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 a useful debugging aid. 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 293

“Compiler Message Format” on page 294

“flag” on page 94

“info” on page 114

“langlvl” on page 134

“source” on page 200

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

For information about environment variables used by the XL C/C++ Advanced Edition for Mac OS X compiler, see the following topics in this section:

- “Set Environment Variables”
- “Set Other Environment Variables”

Set Environment Variables

The Bourne Again SHell (**bash**) 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++ Advanced Edition for Mac OS X 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:

```
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 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.
MANPATH	Specifies the directory search path for finding man pages. MANPATH must contain /opt/ibmcmp/vacpp/6.0/man/en_US/man before the default man path.
DYLIB_LIBRARY_PATH	Specifies the directory search path for dynamically loaded libraries. Used by the GNU linker at link time and at run time.

LANG	<p>Specifies the national language for message and help files.</p> <p>The LANG environment variable can be set to any of the locales provided on the system.</p> <p>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.</p> <p>To determine the current setting of the national language on your system, use the both of the following echo commands:</p> <pre>echo \$LANG echo \$NLSPATH</pre>
NLSPATH	<p>Specifies the path name of the message and help files.</p>
PDFDIR	<p>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.</p>
TMPDIR	<p>Specifies the directory in which temporary files are created. The default location, <code>/tmp</code>, may be inadequate at high levels of optimization, where paging and temporary files can require significant amounts of disk space.</p>

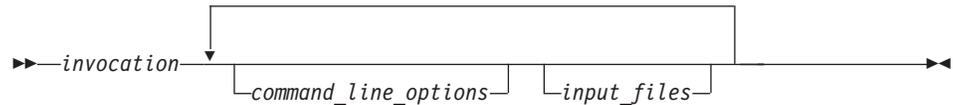
Note: The **LANG** and **NLSPATH** environment variables are initialized when the operating system is installed, and might differ from the ones you want to use.

Related Tasks

“Set Environment Variables” on page 11

Invoke the Compiler

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



The parameters of the compiler invocation command can be the names of input files, compiler options, and linkage-editor 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.

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 invocation command, specifying the object files without the **-c** option.

Notes:

1. Any object files produced from an earlier compilation are deleted as part of the compilation process, even if new object files are not produced.
2. 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**, **-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 **.dylib** suffix, for example, **libold.dylib**.

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 **-ofile_name** option with the **xlc++** command, where *file_name* is the name you want to give to the executable file. If you use the **-qmkshrobj** option to create a shared library, the default name of the shared object created is **shr.o**.

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 `.o` files.

Related Concepts

“Compiler Modes” on page 3

Related Tasks

“Specify Compiler Options” on page 15

“Invoke the Compiler” on page 13

Related References

“Compiler Command Line Options” on page 35

“Message Severity Levels and Compiler Response” on page 293

Appendix C, “Libraries in XL C/C++ Advanced Edition for Mac OS X,” on page 311

Specify Compiler Options

You can specify compiler options in one or more of three ways:

- On the command line (see page 15)
- In your source program (see page 17)
- In a configuration (.cfg) file (see page 17)

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

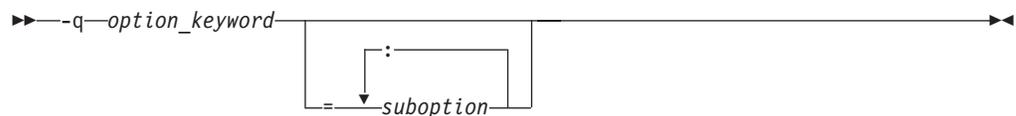
Specify Compiler Options on the Command Line

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 options set 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:

- `-qoption_keyword` (compiler-specific)
- Flag options

-q Options



Command-line options in the `-qoption_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 `-qoption_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 `-qoption_keyword` before or after the file name. For example:

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

Some options have suboptions. You specify these with an equal sign following the `-qoption`. 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 suboptions `w` (warning) for the minimum level of severity to be reported on the listing, and `e` (error) for the minimum level of severity to be reported on the terminal. The option **-qattr** with suboption `full` will produce an attribute listing of all identifiers in the program.

Flag Options

The compilers available on systems use a number of common conventional flag options. IBM XL C/C++ Advanced Edition for Mac OS X 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++ Advanced Edition for Mac OS X also supports flags directed to other programming tools and utilities (for example, the `ld` command). The compiler passes on those flags directed to `ld` at link-edit time.

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

```
xlc stem.c -F/home/tools/test3/new.cfg:myc -qproclcal=sort:count
```

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

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

```
xlc -0cv file.c
```

has the same effect as:

```
xlc -0 -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 -0vo test test.c
```

has the same effect as:

```
xlc -0 -v -otest test.c
```

Most flag options are a single letter, but some are two letters. Note that `-pg` (extended profiling) is not the same as `-p -g` (profiling, `-p`, and generating debug information, `-g`). 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 4

Related Tasks

“Invoke the Compiler” on page 13

“Specify Compiler Options in Your Program Source Files” on page 17

“Specify Compiler Options in a Configuration File”
“Specify Compiler Options for Architecture-Specific Compilation” on page 19
“Resolving Conflicting Compiler Options” on page 19

Related References

“Compiler Command Line Options” on page 35

See also the *GNU C and C++ to XL C/C++ option mapping* section of the *Getting Started with XL C/C++ for Mac OS X*.

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 giving a 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.

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 4

Related Tasks

“Invoke the Compiler” on page 13

“Specify Compiler Options on the Command Line” on page 15

“Specify Compiler Options in a Configuration File”

“Specify Compiler Options for Architecture-Specific Compilation” on page 19

“Resolving Conflicting Compiler Options” on page 19

Related References

“General Purpose Pragmas” on page 240

Specify Compiler Options in a Configuration File

The default configuration file (`/etc/opt/ibmcomp/vac/6.0/vac.cfg`) defines values and compiler options for the 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 is installed to `/etc/opt/ibmcomp/vac/6.0/vac.cfg`.

Before using the compiler for the first time, you must use the `new_install` utility to create your own configuration file based on the template file. By default, the `new_install` utility creates the new configuration file at `/etc/opt/ibmcomp/vac/6.0/vac.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 *Configure the compiler* 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:

Related Concepts

“Compiler Options” on page 4

Related Tasks

“Invoke the Compiler” on page 13

“Specify Compiler Options on the Command Line” on page 15

“Specify Compiler Options in Your Program Source Files” on page 17

“Specify Compiler Options for Architecture-Specific Compilation” on page 19

“Resolving Conflicting Compiler Options” on page 19

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 Mac OS X*.

Specify Compiler Options for Architecture-Specific Compilation

You can use IBM XL C/C++ Advanced Edition for Mac OS X compiler options to optimize compiler output for use on specific processor architectures.

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

1. Configuration file settings
2. Command line compiler options (**-qarch**, **-qtune**)
3. Source file statements (**#pragma options tune=suboption**)

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

- *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 **ppcv**.
- 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.

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

- **-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** to **ppcv**, and sets **-qtune** to the **-qarch** setting's default **-qtune** setting.

Related Concepts

"Compiler Options" on page 4

Related Tasks

"Invoke the Compiler" on page 13

"Specify Compiler Options on the Command Line" on page 15

Related References

"Compiler Command Line Options" on page 35

"Resolving Conflicting Compiler Options"

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.

If a command-line flag is valid for more than one compiler program (for example **-B**, **-W**, or **-I** applied to the compiler, linkage editor, and assembler program names), you must specify it in **codeopt**, **inlineopt**, **ldopt**, or **asopt** in the

configuration file. The command-line flags must appear in the order that they are to be directed to the appropriate compiler program.

Two exceptions to the rules of conflicting options are the *-Idirectory* and *-Ldirectory* 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 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.

Most options that do not follow this scheme are summarized in the following table.

Option	Conflicting Options	Resolution
-qhalt	Severity specified	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	-E
-P	-c -o	-P
-#	-v	-#
-F	-B -t -W -qpath configuration file settings	-B -t -W -qpath
-qpath	-B -t	-qpath overrides -B and -t

Related Concepts

“Compiler Options” on page 4

Related Tasks

“Invoke the Compiler” on page 13

“Specify Compiler Options on the Command Line” on page 15

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 **-Idirectory** 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++ Advanced Edition for Mac OS X supports both. With the **#include** directive, you can search directories by enclosing the file name between < > or " " characters.

The result of using each method is as follows:

#include type	Directory Search Order
#include <file_name>	<ol style="list-style-type: none">1. If you specify the -Idirectory option, the compiler searches for <i>file_name</i> in the directory called <i>directory</i> first. If more than one directory is specified, the compiler searches the directories in the order that they appear on the command line.2. For C++ compilations, the compiler searches the directories specified by the -qcpp_stdinc and -qgcc_cpp_stdinc compiler options.3. The compiler searches the directories specified by the -qc_stdinc and -qgcc_c_stdinc compiler options.4. If -qstdframework is in effect, the compiler searches for frameworks in system and user-specified framework directories.

#include <i>"file_name"</i>	<ol style="list-style-type: none"> 1. Starts searching from the directory where your current source file resides. The current source file is the file that contains the directive #include <i>"file_name"</i>. 2. If you specify the option -Idirectory, the compiler searches for <i>file_name</i> in <i>directory</i>. If more than one directory is specified, the compiler searches the directories in the order that they appear on the command line. 3. For C++ compilations, the compiler searches the directories specified by the -qcpp_stdinc and -qgcc_cpp_stdinc compiler options. 4. The compiler searches the directories specified by the -qc_stdinc and -qgcc_c_stdinc compiler options. 5. If -qstdframework is in effect, the compiler searches for frameworks in system and user-specified framework directories.
------------------------------------	--

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 **-Idirectory** option.

Use the **-qnostdinc** option to search only the directories specified with the **-Idirectory** 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 **-Idirectory** option before searching other directories.

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

Related References

"I" on page 110
"c_stdinc" on page 65
"cpp_stdinc" on page 80
"framework" on page 100
"frameworkdir" on page 101
"gcc_c_stdinc" on page 104
"gcc_cpp_stdinc" on page 105
"stdframework" on page 206

Structure a Program that Uses Templates

The following class template, `Stack`, is used as an example in the sections that follow. `Stack` implements a stack of items. The overloaded operators `<<` and `>>` are used to push items to the stack and pop items from the stack. Both return an integer result: 1 = success, 0 = failure. The declaration of the `Stack` class template is contained in the file `stack.h`

See the following sections:

- “Declaration of `Stack` in `stack.h`”
- “Declaration of operator Functions in `stack.c`” on page 24
- “Template Functions Declared Inline and Template Functions With Internal Linkage” on page 24
- “Template Functions Defined within the Compilation Unit” on page 25
- “Generate Template Functions Automatically” on page 28
- “Define Template Functions Directly in Compilation Units” on page 27
- “Use the Template Registry to Define Template Functions” on page 26

See also the *Templates* discussion in the *C/C++ Language Reference*.

Declaration of `Stack` in `stack.h`

```
typedef enum{tr,fl} Bool;
template <class Item, int size> class Stack {
public:
    int operator << (Item item); // Push operator
    int operator >> (Item& item); // Pop operator
    Stack(Bool p=f1) {top = 0;} // Constructor defined inline
private:
    Item stack[size]; // The stack of elements
    int top; // Index to top of stack
};
```

In this example, the constructor function is defined inline, and has external linkage. The other functions are defined using separate function templates. These members of class template are contained out of line in the file `stack.c`

Related Tasks

- “Structure a Program that Uses Templates”
- “Declaration of operator Functions in `stack.c`” on page 24
- “Template Functions Declared Inline and Template Functions With Internal Linkage” on page 24
- “Template Functions Defined within the Compilation Unit” on page 25
- “Generate Template Functions Automatically” on page 28
- “Define Template Functions Directly in Compilation Units” on page 27
- “Use the Template Registry to Define Template Functions” on page 26

Related References

- “tempinc” on page 215

Declaration of operator Functions in stack.c

```
template <class Item, int size>
int Stack<Item,size>::operator << (Item item) {
    if (top >= size) return 0;
    stack[top++] = item;
    return 1;
}
template <class Item, int size>
int Stack<Item,size>::operator >> (Item& item)
{
    if (top <= 0) return 0;
    item = stack[--top];
    return 1;
}
```

Related Tasks

“Structure a Program that Uses Templates” on page 23

“Declaration of Stack in stack.h” on page 23

“Template Functions Declared Inline and Template Functions With Internal Linkage”

“Template Functions Defined within the Compilation Unit” on page 25

“Generate Template Functions Automatically” on page 28

“Define Template Functions Directly in Compilation Units” on page 27

“Use the Template Registry to Define Template Functions” on page 26

Related References

“tempinc” on page 215

Template Functions Declared Inline and Template Functions With Internal Linkage

A template function is considered to be *inline* if one of the following applies:

- it is defined within a class definition
- it is declared using the `inline` specifier

An inline function is defined in each translation unit in which it is used and has exactly the same definition in each case. Thus, the compiler generates the same function in each of the compilation units where the template function is instantiated. The compiler may also inline the function for you (inline substitution of the function body at the point of call, similar to macro substitution).

A namespace scope template function has internal linkage if it is explicitly declared `static`. No other template function has internal linkage. The `inline` function specifier does not affect the linkage of a template function. You must define a template function that has internal linkage within the compilation unit in which it is used (implicitly or explicitly instantiated) because a name that has internal linkage cannot be referred to by other names from other translation units.

The definition of a template function must be in scope (visible) at the point of instantiation. On the other hand, the only requirement for a template function to be implicitly instantiated, is that the function declaration has to be in scope at the point of instantiation.

In the Stack template class example, the constructor is defined inline in the class template declaration. As a result, any compilation unit that uses an instance of the Stack class will have the appropriate constructor generated as an inline function by the compiler.

Related Tasks

- “Structure a Program that Uses Templates” on page 23
- “Declaration of Stack in stack.h” on page 23
- “Declaration of operator Functions in stack.c” on page 24
- “Template Functions Defined within the Compilation Unit”
- “Generate Template Functions Automatically” on page 28
- “Define Template Functions Directly in Compilation Units” on page 27
- “Use the Template Registry to Define Template Functions” on page 26

Related References

- “tempinc” on page 215

Template Functions Defined within the Compilation Unit

If a compilation unit explicitly instantiates or specializes a template function or static data member of a template with a set of arguments, the compiler generates the definition. At link-edit time, references in all compilation units to this function or static data member are resolved to this definition. If different compilation units try to explicitly instantiate or specialize the same template function or static data member with the same set of arguments, the compiler may issue messages warning of duplicate symbol definitions.

If a compilation unit contains a template declaration that defines a function or static data member of a template, the compiler generates the code for all functions and static data members that are explicitly specialized or implicitly instantiated within the compilation unit, and for which the definition of the template function or static data member is in scope at the point of instantiation

More than one compilation unit could meet this criteria. If so, several compilation units may generate code for the same template function or static data member of a template.

The link-edit step does not remove any unused template function or static data member code from the executable program. Therefore, if the same code that defines functions or static data members is contained in multiple compilation units, you may generate a very large executable program. In the Stack class template example, for any compilation units that include the file `stack.c`, the compiler generates code for each Stack class instance in that compilation unit. For example, a compilation unit that contains:

```
#include "stack.h"
#include "stack.c"
void Swap(int &i, Stack<int,20>& s)
{
    int j;
    s >> j;
    s << j;
    i = j;
}
```

will automatically generate code for these functions :

```
Stack<int,20>::operator << (int)
Stack<int,20>::operator >> (int&)
```

Related Tasks

- “Structure a Program that Uses Templates” on page 23
- “Declaration of Stack in stack.h” on page 23
- “Declaration of operator Functions in stack.c” on page 24
- “Template Functions Declared Inline and Template Functions With Internal

Linkage” on page 24

“Generate Template Functions Automatically” on page 28

“Define Template Functions Directly in Compilation Units” on page 27

“Use the Template Registry to Define Template Functions”

Related References

“tempinc” on page 215

Use the Template Registry to Define Template Functions

The **-qtemplateregistry** option does not impose specific requirements on the organization of your source code. Any program that compiles successfully when both **-qnotempinc** and **-qnotemplateregistry** are in effect (i.e., the *instantiate at every occurrence* approach) will also compile when **-qtemplateregistry** is in effect.

The **-qtemplateregistry** option relies on a *first-come, first-serve* type of algorithm. When a program references a new instantiation for the first time, it is instantiated in the compilation in which it occurs. When another compilation unit references the same instantiation, it is not instantiated again. Thus, only one copy is generated for the entire program. The information to accomplish this is stored in a template registry file, and you must use the same registry file for the entire program. The default file name for the template registry file is **templateregistry**, but you can use the **-qtemplateregistry** compiler option to specify any other valid file name to override this default. When cleaning your program build environment before starting a fresh or scratch build, you must delete the registry file along with the old object files.

-qtemplateregistry must not be used together with **-qtempinc**. Before initiating a build that uses the **-qtemplateregistry** option, ensure that there are no instantiation files in subdirectory `tempinc` of your working directory.

Recompiling Parts of Your Program After Making Source Changes

If you change your source code and recompile only the affected parts, you could possibly change the dependencies between compilation units. The template registry handles this automatically because it stores all references to templates as well as all the instantiations.

For example, if you have compilation units A and B that both reference the same instantiation and you compile A first, then A’s object file will contain the code for the instantiation. If you modify A so that it no longer references the instantiation, and you recompile A only, then its object file will no longer contain the code for the instantiation. Without further action an undefined symbol error will occur. To handle this situation the compiler will automatically recompile B using the same compiler options as it did for A, so that B’s object file contains the code for the instantiation.

If necessary, automatic recompilation of dependent compilation units can be disabled with the **-qnotemplatercompile** compiler option.

Related Tasks

“Structure a Program that Uses Templates” on page 23

“Declaration of Stack in `stack.h`” on page 23

“Declaration of operator Functions in `stack.c`” on page 24

“Template Functions Declared Inline and Template Functions With Internal Linkage” on page 24

[“Template Functions Defined within the Compilation Unit”](#) on page 25
[“Generate Template Functions Automatically”](#) on page 28
[“Define Template Functions Directly in Compilation Units”](#)

Related References

[“tempinc”](#) on page 215
[“templaterecompile”](#) on page 216
[“templateregistry”](#) on page 217

Define Template Functions Directly in Compilation Units

You can structure your program to define the template functions directly in your compilation units. If you know what instances of a particular template function will be needed, you can define both the template functions and the necessary declarations in one compilation unit. If you use this method, you do not have to reference any compiler-generated files.

However, if you change the body of the function template, you may have to recompile many of the files. Compile and link time may be longer, and the object file produced may become quite large. Careful program structuring can avoid these issues.

To structure your program without using automatic template generation:

1. Ensure that the **-qnotempinc** and **-qnotemplateregistry** compiler options are in effect so that the compiler does not generate tempinc or template registry files.
2. Place the template function definitions into one or more of your compilation units.
3. Place a reference for each template function to be generated in a compilation unit that also contains a definition of the function. Code is generated if the template definition is visible and there is an implicit or explicit instantiation, or if you write a new definition for specific template arguments via an explicit specialization. There is no distinction between member and non-member functions or static data members for this.

In the Stack class template example above, the compiler generates the necessary function code if you include both **stack.h** and **stack.c** in all compilation units which use instances of the Stack class. Code is generated for all the necessary functions. Code may be generated multiple times resulting in a very large object file.

If the tempinc feature is on, the macro `__TEMPINC__` is defined in all compilation units in which automatic template generation is on. This allows you to write code that will compile correctly with and without the **-qtempinc** option by using conditional compilation.

Related Tasks

[“Structure a Program that Uses Templates”](#) on page 23
[“Declaration of Stack in stack.h”](#) on page 23
[“Declaration of operator Functions in stack.c”](#) on page 24
[“Template Functions Declared Inline and Template Functions With Internal Linkage”](#) on page 24
[“Template Functions Defined within the Compilation Unit”](#) on page 25
[“Generate Template Functions Automatically”](#) on page 28
[“Use the Template Registry to Define Template Functions”](#) on page 26

Related References

“tempinc” on page 215

Generate Template Functions Automatically

To avoid producing template code for the same function multiple times, use the compiler to automatically generate the template functions.

The compiler can generate template function code automatically, provided the template functions are referenced but not defined in your program code. To use this method, you must generate a special file called a tempinc file that the compiler uses to generate the function code.

With this tempinc file, the compiler generates each function definition only once for the whole program. The compiler determines what instances of the function must be created and avoids generating multiple copies of the template functions.

You can specify more than one tempinc file for a header file using the **#pragma implementation** directive.

To generate template functions automatically:

1. Declare but do not define the template function.
2. Place the class or function template declaration in a header file and include this header file in your source program by using the `#include` directive. If the template function has class (template) scope, its declaration is part of the class (template) definition. If the template function has namespace scope, you must declare but not define the function using a function template.
3. Create a special tempinc file for each of the header files that contain these template declarations. Use the same name for the tempinc file as for the header file but use a `.c` (lower case c) instead of a `.h` suffix. Place these tempinc files in the same directories as their correspondent `.h` files.
4. Define all the functions declared in the header file in this tempinc file.
5. Place the definitions of any types (classes) that are used in template arguments in header files (so with other words the types used for implicit instantiation). If the class definitions require other header files, use the `#include` directive to include them in either your implementation file or in the `.h` file (not in the main file that uses the template). If any type (class) is required to declare a template function (for example, the types are used as parameter types), place them either in the template declaration file (the `.h` file) or in a separate header file. If you use a header file, include it in the template declaration file using the `#include` directive and NOT directly in your main file (for example the `Bool` type in the stack example). Do not put the definitions of any classes that are used in template arguments (the implicit instantiation) or in template function definition in your source file. If a user-defined type is used in an implicit function instantiation, the compiler will automatically include the file in the tempinc generated template file, but will not do that for types used in template function definition or declaration.
6. If you compile and then link at a different time, repeat any compiler options you specified at compile time, when you link. Using the same compiler options allows the compiler to properly compile the template-instantiation files that are generated at compile time. For example, use the same path names for the `-Idirectory` option so that the compiler uses the same include files.

Note: If you have many files that all use the same template with the same arguments, do not use the `-qnotempinc` option. Automatic function generation is disabled by the `-qnotempinc` option. In the Stack class template example, the `stack.h` header file is included in any compilation units that use instances of the class. The `stack.c` file is not included by any of these compilation units. The compiler uses it to build the necessary functions. For example, a main compilation unit may contain:

```
#include "stack.h"
void Swap(int &i, Stack<int,20>& s)
{
    int j;
    s >> j;
    s << j;
    i = j;
}
```

During the link-edit step, the compiler will automatically generate code for these functions :

```
Stack<int,20>::operator << (int)
Stack<int,20>::operator >> (int&)
```

How the Compiler Generates the Function Definitions

During compilation of your program, the compiler builds up a special template instantiation file for each header file that contains template functions or static data members of a template class that require code generation. The compiler stores these template instantiation files in the `tempinc` subdirectory of the working directory. It creates the `tempinc` subdirectory if one does not already exist.

Before link-editing your program, the compiler checks the contents of the `tempinc` subdirectory, compiles its template instantiation files, and generates the necessary template code.

You can rename the `tempinc` file or place it in a different directory with the **`#pragma implementation`** directive. If you designate a relative path name, the path must be relative to the directory containing the header file.

In the Stack class template example, if you want to use the file `stack.defs` as the `tempinc` file instead of `stack.c`, add the line `#pragma implementation("stack.defs")` anywhere in the `stack.h` file. The compiler expects to find the `tempinc` file `stack.defs` in the same directory as `stack.h`.

Specifying the tempinc file

Define all the functions declared in the header file in the `tempinc` files. These definitions can be explicit specializations, template definitions, or both. Static data members must be defined too. If you include explicit specializations, the compiler uses them rather than those generated from the template when it processes the template instantiation file.

If you use a class as a template argument and the class definition is needed in the `tempinc` file to generate the template function, include the class definition in the header file. The compiler includes the header file in the template instantiation file. This makes the class definition available when the function definition is compiled.

Specifying a Different Path for the tempinc Subdirectory

By default, the compiler builds and compiles the special template-instantiation files in the **tempinc** subdirectory of the working directory. To redirect these files to another directory, use the **-qtempinc** option. If you specify a directory, make sure you specify it consistently for all compilations and link-edits of your program.

Regenerating the Template Instantiation File

The compiler builds a template instantiation file corresponding to each header file containing template function declarations. After the compiler creates one of these files, it may add information to it as each compilation unit is compiled. The compiler never removes information from the file.

As you develop your program, you may remove function instantiations or reorganize your program, so that the template instantiation files become obsolete. Since the compiler does not remove information from the template instantiation files, you may want to delete one or more of these files and recompile your program periodically. To regenerate all of the template instantiation files, delete the tempinc directory and recompile your program.

If a compiler-generated file in a tempinc directory has compiler errors, the file will be recompiled whenever a link is done using the **xlc++** compiler invocation with that tempinc directory. To avoid this recompilation, either fix the errors in the file or remove it from the tempinc directory.

Breaking a Template Instantiation File into More Than One File

Normally the compiler generates one template instantiation file for each template header file. When compiled, the template instantiation file may be too large to be created by the compiler. If so, you can break the template instantiation file into more than one file using the **-qtempmax=number** compiler option. A hash-type algorithm is used to split up the instantiations, so you should use an appropriately-sized prime number as the best choice for *number*. More than one object file will be created, all of them smaller in size than the first one.

Contents of Template Instantiation File

This section contains two examples of template instantiation files. The first is an example showing the information that would be in a typical template instantiation file. The second is the file produced for Stack class template example.

Example of a Typical Template-Instantiation File

The following example shows the layout of a typical template instantiation file generated by the compiler. The compiler does not remove information from the file. You can edit these files but it is neither necessary nor advisable.

```
/*0698421265*/#include "/home/myapp/src/list.h"      1
/*0000000000*/#include "/home/myapp/src/list.c"      2
/*0698414046*/#include "/home/myapp/src/mytype.h"    3
/*0698414046*/#include "/usr/vacpp/include/iostream.h" 4
template int List<MyType>::foo();                    5
template ostream& operator<<(ostream&, List<MyType>); 6
template int count(List<MyType>); 1. 2. 3. 4.        7
```

A descriptions of each line follows:

1. The header file that corresponds to the template-instantiation file. The number in comments at the start of each **#include** line (in this case, **/*0698421265*/**) is a timestamp for the included file. The compiler uses this number to determine if

the template-instantiation file should be recompiled. A time stamp of zeroes (as in line 2.) means the compiler is to ignore the timestamp.

2. The tempinc file that corresponds to the header file in line 1.
3. Another header file that the compiler needs to compile the template-instantiation file. All other header files that the compiler needs to compile the template-instantiation file are inserted at this point. In this example, the type MyType is used as a template argument and was defined in the mytype.h header file (MyType is needed for the instantiation of template function foo).
4. Another include file inserted by the compiler. It is referenced in the function explicit instantiation at line 6 (ostream is needed for the instantiation of the operator <<).
5. Code for the member function foo of class template List is going to be generated, for the specific type MyType, by this explicit instantiation.
6. The operator << function has namespace scope. It is matching a template declaration in the file list.h and has its definition in list.c. The compiler inserted this explicit instantiation to force the generation of the function code.
7. count function is a function that has namespace scope. The compiler inserted this explicit instantiation to force the generation of the function code.

Template-Instantiation File (Stack.C) for the Stack class Template Example

```
/*0709395703*/#include "/home/myapp/stack.h"  
/*00000000000*/#include "/home/myapp/stack.c"  
template int Stack<int,20>::operator<<(int);  
template int Stack<int,20>::operator>>(int &);
```

Using #pragma Directives in Header Files

When a #pragma directive is specified in a program, the directive is in effect until it is reset or overridden. You must reset any #pragma directives that would have an unwanted effect on other include files. For example, if you had a header file, **header1.h**, with:

```
...  
#pragma options enum=small  
enum enum1 {p,q,r,s};  
...
```

and another file, **header2.h**, with:

```
...  
enum enum2 {a,b,c,d};  
...
```

enum2 would be treated as small if **header2.h** followed **header1.h**. enum2 would be treated as int if **header2.h** preceded **header1.h** and if **header2.h** had no #pragma options enum=small directive. In this example, you should specify #pragma options enum=reset at the end of **header1.h** to avoid any carry over to another file.

Considerations for Shared Libraries

In a traditional application development environment, different applications can share both source files and compiled files. If you decide to use templates, applications can share source files but cannot share compiled files.

If you use templates:

- Each application must have its own template directory.
- You must compile all of the files for the application, even if some of the files have already been compiled for another application.

Related Tasks

“Structure a Program that Uses Templates” on page 23

“Declaration of Stack in stack.h” on page 23

“Declaration of operator Functions in stack.c” on page 24

“Template Functions Declared Inline and Template Functions With Internal Linkage” on page 24

“Template Functions Defined within the Compilation Unit” on page 25

“Define Template Functions Directly in Compilation Units” on page 27

“Use the Template Registry to Define Template Functions” on page 26

Related References

“tempinc” on page 215

Part 3. Reference

Compiler Options

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

- “Compiler Command Line Options”
- “General Purpose Pragmas” on page 240

Compiler Command Line Options

This section lists and describes XL C/C++ Advanced Edition for Mac OS X 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.

Options that appear entirely in lowercase must be entered in full.

Option Name	Type	Default	Description
+ (plus sign)	<i>-flag</i>	-	C++ Compiles any file, <i>filename.nnn</i> , as a C++ language file, where <i>nnn</i> is any suffix other than .o , .a , or .s .
# (pound sign)	<i>-flag</i>	-	Traces the compilation without doing anything.
aggrcopy	<i>-qopt</i>	See aggrcopy .	Enables destructive copy operations for structures and unions.
alias	<i>-qopt</i>	See alias .	Specifies which type-based aliasing is to be used during optimization.
align	<i>-qopt</i>	power	Specifies what aggregate alignment rules the compiler uses for file compilation.
alloca	<i>-qopt</i>	-	C Substitutes inline code for calls to function alloca as if #pragma alloca directives are in the source code.
altivec	<i>-qopt</i>	noaltivec	Enables compiler support for AltiVec™ data types.
ansialias	<i>-qopt</i>	See -qansialias .	Specifies whether type-based aliasing is to be used during optimization.

Option Name	Type	Default	Description
arch	<i>-qopt</i>	arch=ppcv	Specifies the architecture on which the executable program will be run.
assert	<i>-qopt</i>	noassert	C Instructs the compiler to apply aliasing assertions to your compilation unit.
attr	<i>-qopt</i>	noattr	Produces a compiler listing that includes an attribute listing for all identifiers.
B	<i>-flag</i>	-	Determines substitute path names for the compiler, assembler, linkage editor, and preprocessor.
bitfields	<i>-qopt</i>	bitfields=signed	Specifies if bitfields are signed.
bundle	<i>-flag</i>	-	Instructs the linker to create a bundle.
bundle_loader	<i>-flag</i>	-	Specifies the name of a bundle loader program.
C	<i>-flag</i>	-	Preserves comments in preprocessed output.
c	<i>-flag</i>	-	Instructs the compiler to pass source files to the compiler only.
c_stdinc	<i>-qopt</i>	-	C Changes the standard search location for the C headers.
cache	<i>-qopt</i>	-	Specify a cache configuration for a specific execution machine.
chars	<i>-qopt</i>	chars=signed	Instructs the compiler to treat all variables of type <code>char</code> as either signed or unsigned .
check	<i>-qopt</i>	nocheck	Generates code which performs certain types of run-time checking.
cinc	<i>-qopt</i>	nocinc	C++ Include files from specified directories have the tokens extern "C" { inserted before the file, and } appended after the file.
common	<i>-qopt</i>	common	Controls where uninitialized global variables are allocated.
compact	<i>-qopt</i>	nocompact	When used with optimization, reduces code size where possible, at the expense of execution speed.
complexgccincl	<i>-qopt</i>	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	<i>-qopt</i>	See cpluscmt .	C Use this option if you want C++ comments to be recognized in C source files.
cpp_stdinc	<i>-qopt</i>	-	C++ Changes the standard search location for the C++ headers.

Option Name	Type	Default	Description
crt	-qopt	crt	Instructs the linker to use the standard system startup files at link time.
D	-flag	-	Defines the identifier <i>name</i> as in a #define preprocessor directive.
mbsc, dbcs	-qopt	nodbcs	Use the -qdbcs option if your program contains multibyte characters.
dbxextra	-qopt	nodbxextra	-c Specifies that all typedef declarations, struct , union , and enum type definitions are included for debugger processing.
digraph	-qopt	See digraph .	Enables the use of digraph character sequences in your program source.
dollar	-qopt	nodollar	Allows the \$ symbol to be used in the names of identifiers.
E	-flag	-	Instructs the compiler to preprocess the source files.
eh	-qopt	eh	-C++ Controls exception handling.
enum	-qopt	intlong -C++	Specifies the amount of storage occupied by the enumerations.
F	-flag	-	Names an alternative configuration file for the compiler.
flag	-qopt	flag=i:i	Specifies the minimum severity level of diagnostic messages to be reported.
float	-qopt	See float .	Specifies various floating point options to speed up or improve the accuracy of floating point operations.
fltrap	-qopt	nofltrap	Generates extra instructions to detect and trap floating point exceptions.
framework	-flag	-	Names a framework to link to.
frameworkdir	-qopt	See frameworkdir .	Adds a user-defined framework directory to the header file search path.
fullpath	-qopt	nofullpath	Specifies what path information is stored for files when you use the -g option.
g	-flag	-	Generates debugging information for use by a debugger.
gcc_c_stdinc	-qopt	-	-c Changes the standard search location for the gcc headers.
gcc_cpp_stdinc	-qopt	-	-C++ Changes the standard search location for the g++ headers.
genproto	-qopt	nogenproto	-c Produces ANSI prototypes from K&R function definitions.

Option Name	Type	Default	Description
halt	-qopt	C halt=s C++ halt=e	Instructs the compiler to stop after the compilation phase when it encounters errors of specified <i>severity</i> or greater.
haltonmsg	-qopt	-	C++ Instructs the compiler to stop after the compilation phase when it encounters a specific error message.
hot	-qopt	nohot	Instructs the compiler to perform high-order transformations on loops and array language during optimization, and to pad array dimensions and data objects to avoid cache misses.
I	-flag	-	Specifies an additional search path for #include filenames that do not specify an absolute path.
idirfirst	-qopt	noidirfirst	Specifies the search order for files included with the #include "file_name" directive.
ignerrno	-qopt	noignerrno	Allows the compiler to perform optimizations that assume errno is not modified by system calls.
ignprag	-qopt	-	Instructs the compiler to ignore certain pragma statements.
info	-qopt	C noinfo C++ info=lan	Produces informational messages.
initauto	-qopt	noinitauto	Initializes automatic storage to a specified two-digit hexadecimal byte value.
inline	-qopt	See inline.	Attempts to inline functions instead of generating calls to a function.
ipa	-qopt	ipa=object (compile-time) noipa (link-time)	Turns on or customizes a class of optimizations known as interprocedural analysis (IPA).
isolated_call	-qopt	-	Specifies functions in the source file that have no side effects.
keyword	-qopt	See keyword.	Controls whether a specified string is treated as a keyword or an identifier.
L	-flag	See L.	Searches the specified directory at link time for library files specified by the -I option.
l	-flag	See l.	Searches a specified library for linking.
langlvl	-qopt	See langlvl.	Selects the C or C++ language level for compilation.
lib	-qopt	lib	Instructs the compiler to use the standard system libraries at link time.

Option Name	Type	Default	Description
libansi	<i>-qopt</i>	nolibansi	Assumes that all functions with the name of an ANSI C library function are in fact the system functions.
linedebug	<i>-qopt</i>	nolinedebug	Generates abbreviated line number and source file name information for the debugger.
list	<i>-qopt</i>	nolist	Produces a compiler listing that includes an object listing.
listopt	<i>-qopt</i>	nolistopt	Produces a compiler listing that displays all options in effect.
longlong	<i>-qopt</i>	See longlong .	Allows long long types in your program.
M	<i>-flag</i>	-	Creates an output file that contains targets suitable for inclusion in a description file for the make command.
ma	<i>-flag</i>	-	-c Substitutes inline code for calls to function alloca as if #pragma alloca directives are in the source code.
macpstr	<i>-qopt</i>	nomacpstr	Converts Pascal string literals into null-terminated strings where the first byte contains the length of the string.
makedep	<i>-qopt</i>	-	Creates an output file that contains targets suitable for inclusion in a description file for the make command.
maxerr	<i>-qopt</i>	nomaxerr	Instructs the compiler to halt compilation when a specified number of errors of specified or greater severity is reached.
maxmem	<i>-qopt</i>	maxmem=8192	Limits the amount of memory used for local tables of specific, memory-intensive optimizations.
mbscs, dbcs	<i>-qopt</i>	nombcs	Use the -qmbscs option if your program contains multibyte characters.
mkshrobj	<i>-qopt</i>	-	Creates a shared object from generated object files.
O, optimize	<i>-qopt</i> , <i>-flag</i>	nooptimize	Optimizes code at a choice of levels during compilation.
o	<i>-flag</i>	-	Specifies an output location for the object, assembler, or executable files created by the compiler.
P	<i>-flag</i>	-	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	<i>-flag</i>	-	Sets up the object files produced by the compiler for profiling.

Option Name	Type	Default	Description
path	<i>-qopt</i>	-	Constructs alternate program and path names.
pdf1, pdf2	<i>-qopt</i>	nopdf1, nopdf2	Tunes optimizations through Profile-Directed Feedback.
pg	<i>-flag</i>	-	Sets up the object files for profiling.
phsinfo	<i>-qopt</i>	nophsinfo	Reports the time taken in each compilation phase.
pic	<i>-qopt</i>	nopic	Instructs the compiler to generate Position-Independent Code suitable for use in shared libraries.
print	<i>-qopt</i>	print	-qnoprint suppresses listings.
priority	<i>-qopt</i>	-	- C++ Specifies the priority level for the initialization of static constructors
proto	<i>-qopt</i>	noproto	- C Assumes all functions are prototyped.
Q	<i>-flag</i>	See Q .	Attempts to inline functions instead of generating calls to a function.
r	<i>-flag</i>	-	Produces a relocatable object.
report	<i>-qopt</i>	noreport	Instructs the compiler to produce transformation reports that show how program loops are optimized.
ro	<i>-qopt</i>	See ro .	Specifies the storage type for string literals.
roconst	<i>-qopt</i>	See roconst .	Specifies the storage location for constant values.
rtti	<i>-qopt</i>	rtti	- C++ Generates run-time type identification (RTTI) information for the typeid operator and the dynamic_cast operator.
rwvftable	<i>-qopt</i>	norwvftable	- C++ Instructs the compiler to place virtual function tables into read/write memory.
s	<i>-flag</i>	-	Strips symbol table.
showinc	<i>-qopt</i>	noshowinc	Used with -qsource to selectively show user header files (includes using " ") or system header files (includes using < >) in the program source listing.
smallstack	<i>-qopt</i>	nosmallstack	Instructs the compiler to reduce the size of the stack frame.
source	<i>-qopt</i>	nosource	Produces a compiler listing and includes source code.
sourcetype	<i>-qopt</i>	nosourcetype	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	<i>-qopt</i>	spill=512	Specifies the size of the register allocation spill area.

Option Name	Type	Default	Description
srcmsg	<i>-qopt</i>	nosrcmsg	> C Adds the corresponding source code lines to the diagnostic messages in the stderr file.
staticinline	<i>-qopt</i>	nostaticinline	Controls whether inline functions are treated as static or extern.
statsym	<i>-qopt</i>	nostatsym	Adds user-defined, non-external names that have a persistent storage class to the name list.
stdframework	<i>-qopt</i>	stdframework	Determines if system default framework directories are searched for header files.
stdinc	<i>-qopt</i>	stdinc	Specifies which files are included with #include <i><file_name></i> and #include "file_name" directives.
strict	<i>-qopt</i>	See strict .	Turns off aggressive optimizations of the -O3 option that have the potential to alter the semantics of your program.
strict_induction	<i>-qopt</i>	See strict_induction .	Disables loop induction variable optimizations that have the potential to alter the semantics of your program.
suppress	<i>-qopt</i>	See suppress .	Specifies compiler message numbers to be suppressed.
symtab	<i>-qopt</i>	-	Determines what information appears in the symbol table.
syntaxonly	<i>-qopt</i>	-	> C Causes the compiler to perform syntax checking without generating an object file.
t	<i>-flag</i>	See t .	Adds the prefix specified by the -B option to designated programs.
tabsize	<i>-qopt</i>	tabsize=8	Changes the length of tabs as perceived by the compiler.
tempinc	<i>-qopt</i>	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	<i>-qopt</i>	See templaterecompile .	> C++ Helps manage dependencies between compilation units that have been compiled using the -qtemplateregistry compiler option.
templateregistry	<i>-qopt</i>	See 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	<i>-qopt</i>	tempmax=1	> C++ Specifies the maximum number of template include files to be generated by the tempinc option for each header file.

Option Name	Type	Default	Description
threaded	<i>-qopt</i>	See threaded .	Indicates that the program will run in a multi-threaded environment.
tmplparse	<i>-qopt</i>	tmplparse=no	C++ Controls whether parsing and semantic checking are applied to template definition implementations.
trigraph	<i>-qopt</i>	See trigraph .	Enables the use of trigraph character sequences in your program source.
tune	<i>-qopt</i>	See tune .	Specifies the architecture for which the executable program is optimized.
U	<i>-flag</i>	-	Undefines a specified identifier defined by the compiler or by the -D option.
unroll	<i>-qopt</i>	unroll=auto	Unrolls inner loops in the program.
unwind	<i>-qopt</i>	unwind	Informs the compiler that the application does not rely on any program stack unwinding mechanism.
upconv	<i>-qopt</i>	noupconv	C Preserves the unsigned specification when performing integral promotions.
V	<i>-flag</i>	-	Instructs the compiler to report information on the progress of the compilation in a command-like format.
v	<i>-flag</i>	-	Instructs the compiler to report information on the progress of the compilation.
vftable	<i>-qopt</i>	See vftable .	C++ Controls the generation of virtual function tables.
vrsave	<i>-qopt</i>	vrsave	Controls function prolog and epilog code necessary to maintain the VRSAVE register.
W	<i>-flag</i>	-	Passes the listed words to a designated compiler program.
w	<i>-flag</i>	-	Requests that warning messages be suppressed.
warnfourcharconsts	<i>-qopt</i>	nowarnfourcharconsts	Enable warning of four-character constants in program source.
xcall	<i>-qopt</i>	noxcall	Generates code to static routines within a compilation unit as if they were external calls.
xref	<i>-qopt</i>	noxref	Produces a compiler listing that includes a cross-reference listing of all identifiers.
y	<i>-flag</i>	-	Specifies the compile-time rounding mode of constant floating-point expressions.

Option Name	Type	Default	Description
Z	<i>-flag</i>	-	Specifies a search path for library names.

Related Concepts

“Compiler Options” on page 4

Related Tasks

“Specify Compiler Options on the Command Line” on page 15

“Specify Compiler Options in Your Program Source Files” on page 17

“Specify Compiler Options in a Configuration File” on page 17

“Specify Compiler Options for Architecture-Specific Compilation” on page 19

“Resolving Conflicting Compiler Options” on page 19

Related References

“General Purpose Pragmas” on page 240

+ (plus sign)

C++

Purpose

Compiles any file, *filename.nnn*, as a C++ language file, where *nnn* is any suffix other than **.a**, **.dylib**, **.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.cplsp1s` as a C++ source file, enter:

```
xlc++ -+ myprogram.cplsp1s
```

Related References

“Compiler Command Line Options” on page 35

“sourcetype” on page 201

(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

The `-#` option overrides the `-v` option. It displays the same information as `-v`, but does not invoke the compiler. Information is displayed to standard output.

Use this command to determine commands and files 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.

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 230

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=nooverlap** when compiling to the `stdc89`, `extc89`, `stdc99`, `extc99`, `saa`, and `saal2` language levels.

The default setting of this option is **-qaggrcopy=overlap** when compiling to the `EXTENDED` and `CLASSIC` language levels.

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 134

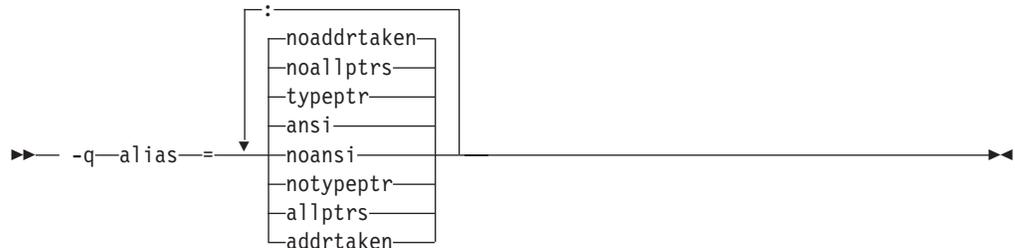
alias

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



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 <i>not</i> 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 <i>only</i> point to an object of the same type. This (ansi) is the default for the xlc , 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 249

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



where available alignment options are:

power	The compiler uses a set of alignment rules to that provide binary compatibility with objects created by gcc V3.3 using the <code>-malign</code> option. This is the default.
natural	The compiler uses a set of alignment rules to that provide binary compatibility with objects created by gcc V3.3 using the <code>-malign</code> option.
mac68k	The compiler uses the Macintosh** alignment rules.
bit_packed	The compiler uses the bit_packed alignment rules. Data, including bitfields, is packed as tightly as possible.

See also “`#pragma align`” on page 242 and “`#pragma options`” on page 272.

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`. 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.

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.

Examples

Example 1 - Affecting Only Aggregate Definition

Using the compiler invocation:

```
xlc++ file2.C /* <-- default alignment rule for file is */
/* power since 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=mac68k file.c /* <-- default alignment rule for file is */
                          /*      Macintosh                          */
xlc -qalign=power file.c /* <-- default alignment rule for file */
                          /*      is power                          */

```

Where file.c has:

```

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

<i>int_const</i>	Specifies a byte-alignment boundary. <i>int_const</i> 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 cannot be used on individual elements within an aggregate definition, but it 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 bytes      */
```

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 242

“#pragma pack” on page 279

See also:

- The *Data Mapping and Storage* section of the *Programming Tasks* manual.
- `__attribute__((aligned))` in the *Variable Attributes* section of the *C/C++ Language Reference*.
- `__attribute__((packed))` in the *Variable Attributes* section of the *C/C++ Language Reference*.

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, or 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
```

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 82

“ma” on page 158

“#pragma alloca” on page 243

altivec

> C > C++

Purpose

Enables compiler support for AltiVec data types.

Syntax

►► -q noaltivec
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.

When this option 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 Programming Interface specification, enter:

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

Related References

“Compiler Command Line Options” on page 35

“#pragma altivec_vrsave” on page 244

ansialias

► C ► C++

Purpose

Specifies whether type-based aliasing is to be used during optimization. Type-based aliasing restricts the lvalues that can be used to access a data object safely.

Syntax

►► -q ansialias
noansialias ◀◀

See also “#pragma options” on page 272.

Notes

This option is obsolete. Use **-qalias=** in your new applications.

The default with **xlC**, **xlC++**, **xlC**, **c99** and **c89** is **ansialias**. The optimizer assumes that pointers can *only* point to an object of the same type.

The default with **cc** is **noansialias**.

This option has no effect unless you also specify the **-O** option.

If you select **noansialias**, 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.

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 -qnoansialias
```

Related References

“Compiler Command Line Options” on page 35

“alias” on page 47

“#pragma options” on page 272

arch

C C++

Purpose

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

Syntax



where available architecture options are:

- auto**
 - Produces object code containing instructions that will run on the hardware platform on which it is compiled.
- g5**
 - Generates instructions specific to G5 processors. This is currently equivalent to specifying **-qarch=ppc970**.
 - Defines the `_ARCH_PPCV` and `_ARCH_G5` macros.
- ppcv**
 - Generates instructions for generic PowerPC chips with AltiVec vector processors. This is the default.
 - Defines the `_ARCH_PPCV` macro.
- ppc970**
 - Generates instructions specific to PowerPC 970 processors.
 - Defines the `_ARCH_PPCV`, `_ARCH_G5`, and `_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.

Example

To specify that the executable program testing compiled from `myprogram.c` is to run on a computer with a PPC970 architecture, enter:

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

Related Tasks

“Specify Compiler Options for Architecture-Specific Compilation” on page 19

Related References

“Compiler Command Line Options” on page 35

“float” on page 95

“O, optimize” on page 171

“tune” on page 223

“Acceptable Compiler Mode and Processor Architecture Combinations” on page 290

assert



Purpose

Requests the compiler to apply aliasing assertions to your compilation unit. The compiler will take advantage of the aliasing assertions to improve optimizations where possible.

Syntax



where available aliasing options include:

noassert	No aliasing assertions are applied.
ASsert=TYPEptr	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.
ASsert=ALLPtrs	Pointers are never aliased (this implies -qassert=typeptr). Therefore, in the compilation unit, no two pointers will point to the same storage location.
ASsert=ADDRtaken	Variables are disjoint from pointers unless their address is taken. Any class of variable for which an address has <i>not</i> been recorded in the compilation unit will be considered disjoint from indirect access through pointers.

See also “#pragma options” on page 272.

Notes

This option is obsolete. Use -qalias= in your new applications.

Related References

“Compiler Command Line Options” on page 35

“alias” on page 47

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++ Advanced Edition for Mac OS X 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++ Advanced Edition for Mac OS X executables and have the option of specifying which one you want to use.

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

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

The **-Bprefix -tprograms** options override the **-Fconfig_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 178

"t" on page 213

bitfields

► C ► C++

Purpose

Specifies if bitfields are signed. By default, bitfields are signed.

Syntax

►► -q-bitfields=
 signed
 unsigned

where options are:

signed	Bitfields are signed.
unsigned	Bitfields are unsigned.

Related References

“Compiler Command Line Options” on page 35

bundle

► C ► C++

Purpose

Instructs the linker to create a bundle.

Syntax

►► -bundle ◀◀

Notes

This option is passed to the linkage editor.

It instructs the linker to create a *bundle* of resources that can be dynamically loaded into an application.

Related References

“Compiler Command Line Options” on page 35

“bundle_loader” on page 62

“framework” on page 100

bundle_loader

► C ► C++

Purpose

Specifies the name of a bundle loader program.

Syntax

►► -bundle_loader—*loader_program_name*◀◀

Notes

This option is passed to the linker, and is used together with the **-bundle** option when creating a *bundle*.

It specifies the name of a bundle loader program that will load the bundle file being created and linked. Undefined symbols in the bundle file are resolved against the bundle loader program.

Example

When creating a bundle, you could use something similar to the following to pass the **-bundle** and **-bundle_loader** options to the linkage editor:

```
xlc++ -bundle -bundle_loader /usr/local/lib/bin/my_lib.bin
```

Related References

“Compiler Command Line Options” on page 35

“bundle” on page 61

“framework” on page 100

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 87

“P” on page 176

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, *file_name.c* or *file_name.i*.

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 `myfile.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 87

“o” on page 175

“P” on page 176

“syntaxonly” on page 212

c_stdinc



Purpose

Changes the standard search location for the C headers.

Syntax



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 21

“Specify Compiler Options in a Configuration File” on page 17

Related References

“Compiler Command Line Options” on page 35

“`cpp_stdinc`” on page 80

“`gcc_c_stdinc`” on page 104

“`stdinc`” on page 207

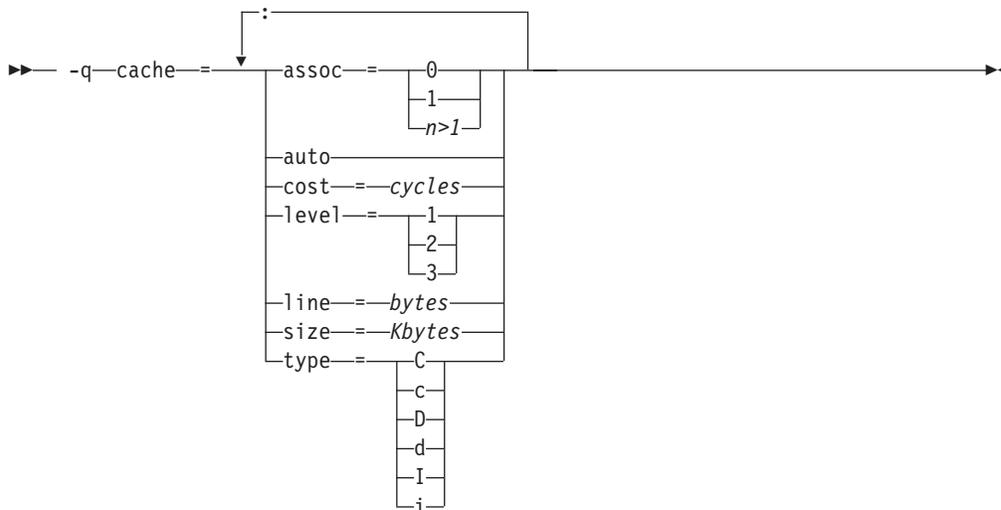
cache

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



where available cache options are:

<code>assoc=number</code>	Specifies the set associativity of the cache, where <i>number</i> is one of: 0 Direct-mapped cache 1 Fully associative cache N>1 n-way set associative cache
<code>auto</code>	Automatically detects the specific cache configuration of the compiling machine. This assumes that the execution environment will be the same as the compilation environment.
<code>cost=cycles</code>	Specifies the performance penalty resulting from a cache miss.
<code>level=level</code>	Specifies the level of cache affected, where <i>level</i> 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 <code>-qcache</code> option.
<code>line=bytes</code>	Specifies the line size of the cache.
<code>size=Kbytes</code>	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 123

“O, optimize” on page 171

chars

C C++

Purpose

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

Syntax

►► -qchars=signed/unsigned ►►

See also “#pragma chars” on page 245 and “#pragma options” on page 272.

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 **unsigned** when compiling myprogram.c, enter:

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

Related References

“Compiler Command Line Options” on page 35

“#pragma chars” on page 245

“#pragma options” on page 272

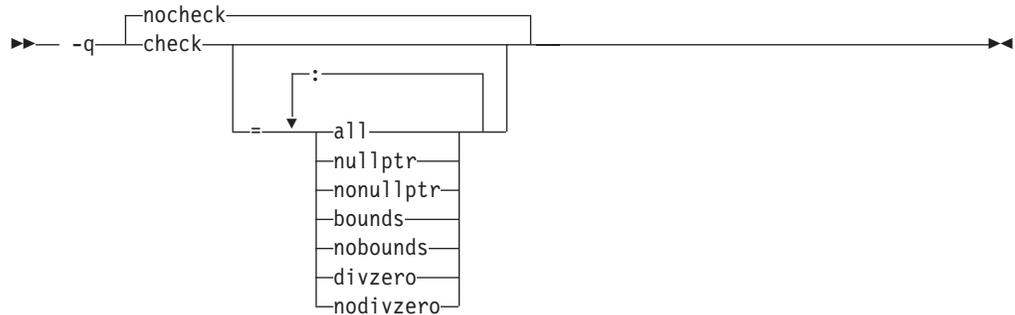
check

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



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:nonnullptr
```

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 | **NONNULLptr**

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.

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 272.

Notes

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

Using the **-qcheck** option without any suboptions turns all 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 **#pragma options** directive must be specified before the first statement in the compilation unit.

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 272

cinc

► C++

Purpose

Include files from specified directories have the tokens **extern "C" {** inserted before the file, and **}** appended after the file.

Syntax

►► -q nocinc
cinc--directory_prefix ►►

where:

<i>directory_prefix</i>	Specifies the directory where files affected by this option are found.
-------------------------	--

Notes

Include files from directories specified by *directory_prefix* have the tokens **extern "C" {** inserted before the file, and **}** appended after the file.

Related References

"Compiler Command Line Options" on page 35

common

C C++

Purpose

Controls where uninitialized global variables are allocated.

Syntax



where:

<code>common</code>	All uninitialized global variables are allocated as common blocks if they are not specified to be allocated in other sections by other options or attributes.
<code>nocommon</code>	All uninitialized global variables, ordinarily allocated as common blocks, are now allocated to the data section of the object file.

Notes

By default, all un-initialized global variables are allocated as common blocks if they are not specified to be allocated in other sections by other options or attributes.

However, when `-qnocommon` is set, all uninitialized global variables will instead be allocated to the data section of the object file.

The `-qnocommon` compiler option is automatically set when `-qmkshrobj` is specified on the command line.

Using [no]common with the `__attribute__` keyword:

You can use the `__attribute__` keyword together with the `common` or `nocommon` specifier to override the current setting of the `-q[no]common` compiler option. Use this keyword/specifier combination to explicitly direct the allocation location for an individual variable or structure.

For example:

```
int i __attribute__((nocommon));    /* allocate i at .data */
int k __attribute__((common));      /* allocate k at .comm */
```

If multiple specifications of `common` or `nocommon` appear in the same attribute statement, the last one specified will take effect. For example:

```
int i __attribute__((common, nocommon));    /* allocate i at .data */
int k __attribute__((common, nocommon, common)); /* allocate k at .comm */
```

The compiler option will not affect the declaration of struct or union members. The compiler will issue a warning message and ignore any attempt to use the `__attribute__` keyword to explicitly assign a `common` or `nocommon` attribute to a struct or union member.

Order of precedence between compiler option and keyword/specifier:

Order of precedence follows these rules:

Compiler option setting	Attribute setting	Section where allocated
Not specified	Not specified	.comm
Not specified	<code>__attribute__((common))</code>	.comm
Not specified	<code>__attribute__((nocommon))</code>	.data
-qcommon	Not specified	.comm
-qcommon	<code>__attribute__((common))</code>	.comm
-qcommon	<code>__attribute__((nocommon))</code>	.data
-qnocommon	Not specified	.data
-qnocommon	<code>__attribute__((common))</code>	.comm
-qnocommon	<code>__attribute__((nocommon))</code>	.data
	Note: Specifying <code>-qnocommon</code> and <code>__attribute__((nocommon))</code> together will prevent global variables from being simultaneously defined in different object files, resulting in an error at link time. Such variables should be defined in one file and referred to in other files with an extern declaration.	

Related References

“Compiler Command Line Options” on page 35

“mkshrobj” on page 167

See also `__attribute__((nocommon))` in the *Variable Attributes* section of the *C/C++ Language Reference*.

compact

► C ► C++

Purpose

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

Syntax

►► -q nocompact
compact ◀◀

See also “#pragma options” on page 272.

Notes

Code size is reduced by inhibiting optimizations that replicate or expand code inline. 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 272

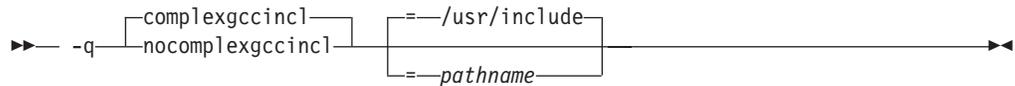
complexgccincl

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



where:

pathname Specifies a search path for include files.

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 95

cplusplus

C

Purpose

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

Syntax

► — -q — nocplusplus
cplusplus —►

Default

The default setting varies:

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

Notes

The **#pragma options** directive must appear before the first statement in the C language source file and applies to the entire file.

The `__C99_CPLUSCMT` compiler macro is defined when **cplusplus** is selected.

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

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 **-qcplusplus** is specified:

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

The correct answer is 2 (2 divided by 1). When **-qcplusplus** 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 ANSI 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 ANSI 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 -qcplusplus
```

Related References

“Compiler Command Line Options” on page 35

“C” on page 63

“E” on page 87

“langlvl” on page 134

“P” on page 176

cpp_stdinc

C++

Purpose

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

Syntax

```
→ -qcpp_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 specify **mypath/headers1** and **mypath/headers2** as being part of 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 21

“Specify Compiler Options in a Configuration File” on page 17

Related References

“Compiler Command Line Options” on page 35

“c_stdinc” on page 65

“gcc_cpp_stdinc” on page 105

“stdinc” on page 207

crt

► C ► C++

Purpose

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

Syntax

►► -q crt
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 150

D

> C > C++

Purpose

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

Syntax

►► -D—*name* _____
 └─┬──────────┘
 └─┬──────────┘
 definition

Notes

The identifier name can also be defined in your source program using the **#define** preprocessor directive.

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

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

To aid in program portability and standards compliance, the provides several header files that define 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 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 224

dbxextra

► C

Purpose

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

Syntax

►► -q nodbxextra
dbxextra _____►►

See also “#pragma options” on page 272.

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 103

“#pragma options” on page 272

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 272.

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:

Keyword	Character Produced
bitand	&
and	&&
bitor	
or	
xor	^
compl	~
and_eq	&=
or_eq	=
xor_eq	^=
not	!
not_eq	!=

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 134

"trigraph" on page 221
"#pragma options" on page 272

dollar

► C ► C++

Purpose

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

Syntax

►► -q nodollar
dollar _____►►

See also “#pragma options” on page 272.

Example

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

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

Related References

“Compiler Command Line Options” on page 35

“#pragma options” on page 272

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** (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 156

“o” on page 175

“P” on page 176

“syntaxonly” on page 212

eh

► C++

Purpose

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

Syntax

► — -q —

eh
noeh

 —————►

Notes

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.

If your program uses C++ exception handling, **-qnoeh** informs the compiler that no C++ exceptions will be thrown, and that cleanup code can be omitted.

Related References

“Compiler Command Line Options” on page 35

“rtti” on page 195

enum

C C++

Purpose

Specifies the amount of storage occupied by enumerations.

Syntax



where valid **enum** settings are:

- 1 Specifies that enumerations occupy 1 byte of storage.
- 2 Specifies that enumerations occupy 2 bytes of storage.
- 4 Specifies that enumerations occupy 4 bytes of storage.
- 8 Specifies that enumerations occupy 8 bytes of storage.
- 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 occupy 8 bytes of storage and are represented by **long long** if the range of the constants exceed the limit for **int**. Otherwise, the enumerations occupy 4 bytes of storage and are represented by **int** or **unsigned 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.

See also “#pragma enum” on page 250 and “#pragma options” on page 272.

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, the requested size is kept but a warning is issued. For example:

```
enum {frog, toad=257} amph;  
1506-387 (W) The enum cannot be packed to the requested size.  
Use a larger value for -qenum.  
(The enum size is 1 and the value of toad is 1)
```

The table below shows 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**.

Enum sizes									
Range of enumeration items	enum=small	enum=1	enum=2	enum=4	enum=int	enum=8	enum=intlong		
0..127	unsigned char	signed char	signed short	signed int	signed int	signed long long	signed int	signed int	enum=intlong
-128..127	signed char	signed char	signed short	signed int	signed int	signed long long	signed int	signed int	signed int
0..255	unsigned char	unsigned char	signed short	signed int	signed int	signed long long	signed int	signed int	signed int
0..32767	unsigned short	ERR ¹	signed short	signed int	signed int	signed long long	signed int	signed int	unsigned int
-32768..32767	signed short	ERR ¹	signed short	signed int	signed int	signed long long	signed int	signed int	signed int
0..65535	unsigned short	ERR ¹	unsigned short	signed int	signed int	signed long long	signed int	signed int	signed int
0..2147483647	unsigned int	ERR ¹	ERR ¹	signed int	signed int	signed long long	signed int	signed int	signed int
-2147483648 ..2147483647	signed int	ERR ¹	ERR ¹	signed int	signed int	signed long long	signed int	signed int	signed int
0..4294967295	unsigned int	ERR ¹	ERR ¹	unsigned int	ERR ¹	signed long long	unsigned int	signed int	unsigned int
0..(2 ⁶³ -1)	signed long long	ERR ¹	ERR ¹	ERR ¹	ERR ¹	signed long long	signed long long	signed long long	signed long long
-2 ⁶³ ..(2 ⁶³ -1)	signed long long	ERR ¹	ERR ¹	ERR ¹	ERR ¹	signed long long	signed long long	signed long long	signed long long
0..2 ⁶⁴	unsigned long long	ERR ¹	ERR ¹	ERR ¹	ERR ¹	unsigned long long	unsigned long long	unsigned long long	unsigned long long

Notes:

1. A severe error message is issued and compilation stopped.

Related References

“Compiler Command Line Options” on page 35

“#pragma enum” on page 250

“#pragma options” on page 272

F

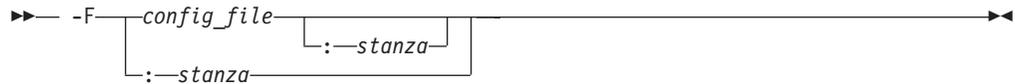
► C ► C++

Purpose

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

Syntax

```
► -F config_file [ :—stanza ]
```



where suboptions are:

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

Notes

The default is a configuration file 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 17.

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 17

Related References

“Compiler Command Line Options” on page 35

“B” on page 59

“t” on page 213

“W” on page 233

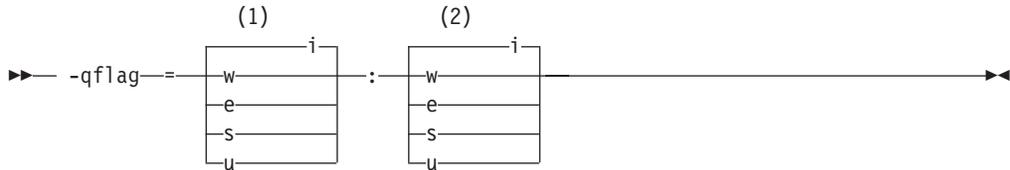
flag

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



Notes:

- 1 Minimum severity level messages reported in listing
- 2 Minimum severity level messages reported on terminal

where message severity levels are:

<i>severity</i>	Description
i	Information
w	Warning
e	Error
s	Severe error
u	Unrecoverable error

See also “#pragma options” on page 272.

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 114

“w” on page 234

“#pragma options” on page 272

“Compiler Messages” on page 293

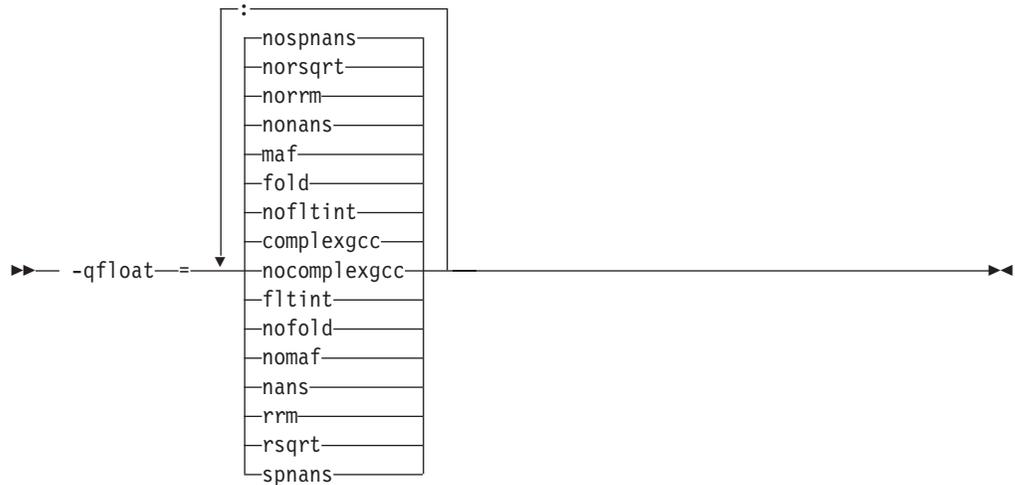
float

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



Option selections are described in the **Notes** section below. See also “#pragma options” on page 272.

Notes

Using the **float** option may produce results that are not precisely the same as the default. Incorrect results may be produced if not all required conditions 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	Enables compatibility with GCC passing and returning of <code>_complex</code> .
nocomplexgcc	The default is float=complexgcc .

fltint nofltint	<p>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.</p> <p>This suboption must only be used with an optimization option.</p> <ul style="list-style-type: none"> • For -O2, the default is -qfloat=nofltint. • For -O3, the default is -qfloat=fltint. <p>To include range checking in floating-point-to-integer conversions with the -O3 option, specify -qfloat=nofltint.</p> <ul style="list-style-type: none"> • -qnostrict sets -qfloat=fltint <p>Changing the optimization level will not change the setting of the fltint suboption if fltint has already been specified.</p> <p>If the -qstrict -qnostrict and -qfloat= options conflict, the last setting is used.</p>
fold nofold	<p>Specifies that constant floating-point expressions are to be evaluated at compile time rather than at run time.</p>
maf nomaf	<p>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. This option may affect the precision of floating-point intermediate results.</p>
nans nonans	<p>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.</p> <p>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.</p>
rrm norrm	<p>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 <i>round to nearest</i> at run time.</p> <p>-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).</p>

`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

“Compiler Command Line Options” on page 35

“arch” on page 56

“complexgccincl” on page 75

“flttrap” on page 98

“strict” on page 208

“#pragma options” on page 272

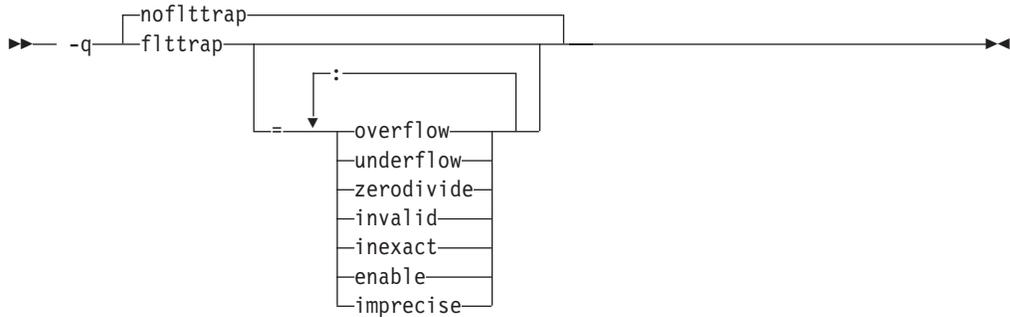
fltrap

C C++

Purpose

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

Syntax



where suboptions do the following:

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.
ENable	Enables the specified exceptions in the prologue of the main program. This suboption is required if you want to turn on exception trapping without modifying the source code.
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.

See also “#pragma options” on page 272.

Notes

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

Specifying the **-qfltrap** option with no suboptions is equivalent to setting **-qfltrap=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 **-qnofltrap** option *must* be the first option specified.

If your program contains signalling NaNs, you should use the **-qfloat=nans** along with **-qfltrap** to trap any exceptions.

The compiler exhibits behavior as illustrated in the following examples when the **-qfltrap** option is specified together with **-qoptimize** options:

- with **-O**:

- 1/0 generates a **div0** exception and has a result of infinity
- 0/0 generates an invalid operation
- with **-O3**:
 - 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 95

“O, optimize” on page 171

framework

C C++

Purpose

Specifies the name of a framework to link to.

Syntax

► — `-framework`—*framework_name* —┬— `—framework_name_ext` —►

Notes

This option is passed to the linkage editor, and specifies the name of a framework to link to.

The compiler searches for the named framework directory *framework_name.framework/framework_name* in the following locations:

1. Directories specified by the `-qframeworkdir` compiler option.
2. `/Library/Frameworks`
3. `/Network/Library/Frameworks`
4. `/System/Library/Frameworks`

If *framework_name_ext* is specified, the compiler first searches for *framework_name.framework_name_ext*, and then if not found, *framework_name* without the extension.

Example

To link to a specific framework library (for example, **Carbon**) during compilation, use the following to pass the framework library name to the linker:

```
xlc++ -framework Carbon -o myprogram myprogram.C
```

Related References

“Compiler Command Line Options” on page 35

“bundle” on page 61

“bundle_loader” on page 62

“frameworkdir” on page 101

“stdframework” on page 206

frameworkdir

C C++

Purpose

Adds a user-defined framework directory to the framework header file search path.

Syntax

```
► -qframeworkdir=directory_path ►
```

Notes

This option passes the specified *directory_path* to the linkage editor's **-F** option.

By default, the compiler will search for a header file in the following locations, listed in order of search priority, until it is found:

1. Ordinary header file locations
2. User-defined framework directories (if specified by the **-qframeworkdir** compiler option)
3. System-default framework directories, listed in order of priority:
 - a. /Library/Frameworks/
 - b. /Network/Library/Frameworks/
 - c. /System/Library/Frameworks/
4. Subframework directories, if in an umbrella framework

User-defined framework directories can be defined with the **-qframeworkdir** compiler option.

Examples

1. The following option specification will add **my_dir1** to the framework header file search path:

```
-qframeworkdir=my_dir1
```

2. The following option specification will add **my_dir1**, **my_dir2**, and **my_dir3** to the framework header file search path:

```
-qframeworkdir=my_dir1 -qframeworkdir=my_dir2 -qframeworkdir=my_dir3
```

User-defined framework directories are searched in the order that they are defined to the compiler. In the above example, **my_dir1** would be searched first, followed by **my_dir2**, and then **my_dir3**.

Related References

“Compiler Command Line Options” on page 35

“framework” on page 100

“stdframework” on page 206

fullpath

> C > C++

Purpose

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

Syntax

►► -q nofullpath
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 103

g

C C++

Purpose

Generates information used for debugging tools.

Syntax

► — -g — ◄

Notes

Avoid using this option with **-O** (optimization) option. The information produced may be incomplete or misleading.

If you specify the **-g** option, the inlining option defaults to **-Q!** (no functions are inlined).

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 testing all containing additional information about unreferenced symbols so you can debug it, enter:

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

Related References

“Compiler Command Line Options” on page 35

“dbxextra” on page 83

“fullpath” on page 102

“linedebug” on page 152

“O, optimize” on page 171

“Q” on page 188

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 21

“Specify Compiler Options in a Configuration File” on page 17

Related References

“Compiler Command Line Options” on page 35

“c_stdinc” on page 65

“cpp_stdinc” on page 80

“gcc_cpp_stdinc” on page 105

“stdinc” on page 207

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 21

“Specify Compiler Options in a Configuration File” on page 17

Related References

“Compiler Command Line Options” on page 35

“c_stdinc” on page 65

“cpp_stdinc” on page 80

“gcc_c_stdinc” on page 104

“stdinc” on page 207

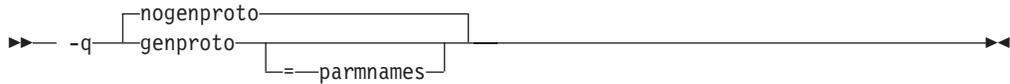
genproto

► C

Purpose

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

Syntax



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;
```

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=parm 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 **chars**, **shorts**, and **floats**) 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



Notes:

- 1 Default for C compilations.
- 2 Default for C++ compilations.

where severity levels in order of increasing severity are:

<i>severity</i>	Description
i	Information
w	Warning
e	Error
s	Severe error
u	Unrecoverable error

See also “#pragma options” on page 272.

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 94

“maxerr” on page 163

“#pragma options” on page 272

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.

Related References

“Compiler Command Line Options” on page 35

“Compiler Messages” on page 293

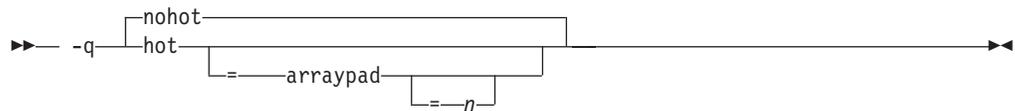
hot

C C++

Purpose

Instructs the compiler to perform high-order transformations on loops and array language during optimization, and to pad array dimensions and data objects to avoid cache misses.

Syntax



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.

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`.

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 171



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

“Compiler Command Line Options” on page 35

Related References

“Directory Search Sequence for Include Files Using Relative Path Names” on page 21

idirfirst

C C++

Purpose

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

Syntax

►► -q noidirfirst
idirfirst ◀◀

See also “#pragma options” on page 272.

Notes

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

With **-qidirfirst**, the directories specified with the **-I** directory 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 option** [NO]IDIRFirst remains in effect until replaced by a subsequent **#pragma option** [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 21

Related References

“Compiler Command Line Options” on page 35

“I” on page 110

“stdinc” on page 207

“#pragma options” on page 272

ignerrno

C C++

Purpose

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

Syntax

►► -q noignerrno
ignerrno ◄◄

See also “#pragma options” on page 272.

Notes

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

Related References

“Compiler Command Line Options” on page 35

“#pragma options” on page 272

ignprag

C C++

Purpose

Instructs the compiler to ignore certain pragma statements.

Syntax

►► -qignprag=
┌ disjoint
├ isolated
└ all

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.

See also “#pragma options” on page 272.

Notes

Suboptions are:

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 directives, enter:

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

Related References

“Compiler Command Line Options” on page 35

“#pragma disjoint” on page 249

“#pragma isolated_call” on page 263

“#pragma options” on page 272

info

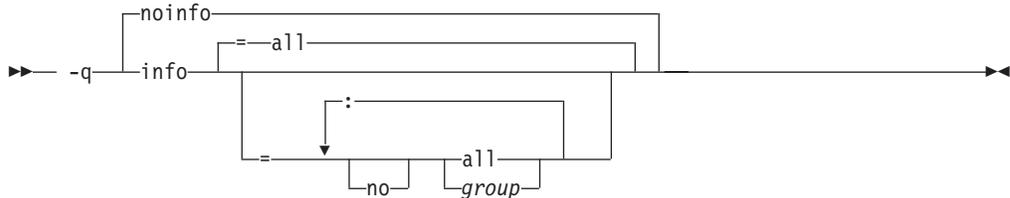
► C ► C++

Purpose

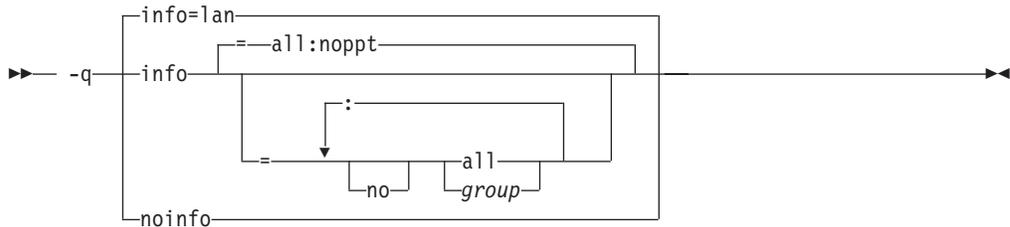
Produces informational messages.

Syntax

► C



► C++



where **-qinfo** options and diagnostic message groups are described in the **Notes** section below. See also “#pragma info” on page 259 and “#pragma options” on page 272.

Defaults

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

1. ► C **-qnoinfo**
2. ► C++ **-qinfo=lan**

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, and **#pragma options info=reset** to return to your initial **-qinfo** settings.

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=all** option does not include the **ppt** group (preprocessor trace).

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

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

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

<i>group</i>	Type of messages returned or suppressed
c99 noc99	C code that may behave differently between C89 and C99 language levels.
c1s noc1s	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
dc1 nodc1	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
use nouse	Unused auto and static variables
vft novft	Generation of virtual function tables

Example

To compile `myprogram.C` to produce informational message about all items except conversions and unreachable statements, enter:

```
xlc++ myprogram.C -qinfo=all -qinfo=nocnv:norea
```

Related References

"Compiler Command Line Options" on page 35

"#pragma info" on page 259

"#pragma options" on page 272

initauto

> C > C++

Purpose

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

Syntax

►► -q noinitauto
initauto=*hex_value* ►►

See also “#pragma options” on page 272.

Notes

The option generates extra code to initialize the automatic (stack-allocated) storage of functions. 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 stack storage is initialized to hex value FF (decimal 255), enter:

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

Related References

“Compiler Command Line Options” on page 35

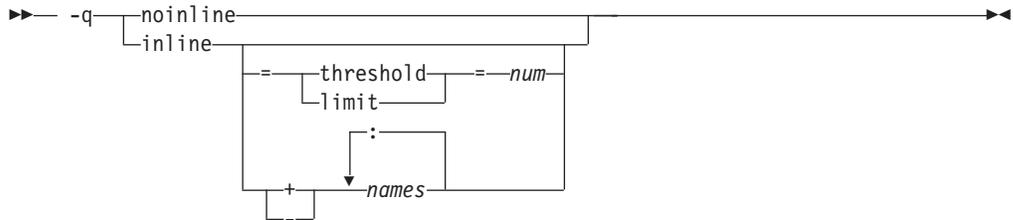
inline

C C++

Purpose

Attempts to inline functions instead of generating calls to a function. 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 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.

<code>-qinline+names</code>	Attempts to inline the functions listed by <i>names</i> and any other appropriate functions. Each <i>name</i> must be separated by a colon (:). The option implies -qinline . For example, <code>-qinline+food:clothes:vacation</code> 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 <i>threshold</i> value. You can use a threshold value of zero along with -qinline+names to inline specific functions. For example: <code>-qinline=threshold=0</code> followed by: <code>-qinline+salary:taxes:benefits</code> causes <i>only</i> the functions named salary , taxes , or benefits to be inlined, if possible, and no others.
<code>-qinline=limit=num</code>	Specifies the maximum size (in bytes of generated code) to which a function can grow due to inlining. This limit does not affect the inlining of user specified functions.
<code>-qnoinline</code>	Does not inline any functions. If -qnoinline is specified last, no functions are inlined.

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

<code>-qinline</code>	Compiler inlines all functions that it can.
<code>-qnoinline</code>	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 specify the **-g** option (to generate debug information), no functions are inlined.
- If you optimize your program **-O**, the compiler attempts to inline all functions declared as inline. Otherwise, the compiler attempts to inline some of the simpler functions declared as inline.

Notes

The **-qinline** option is functionally equivalent to the **-Q** 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++ Advanced Edition for Mac OS X (**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 C compiler provides keywords that you can use to specify functions that you want the compiler to inline:

- `inline`
- `_Inline`
- `_inline`
- `__inline`

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. The inline keywords also implicitly declare the function as static.

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 static 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 specifier keywords, it can be declared without a definition. The following code fragments shows an inline function definition. Note that the definition includes both the declaration and body of the inline function.

```
_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=threshold=12
```

Related References

“Compiler Command Line Options” on page 35

“g” on page 103

“O, optimize” on page 171

“Q” on page 188

ipa

C C++

Purpose

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

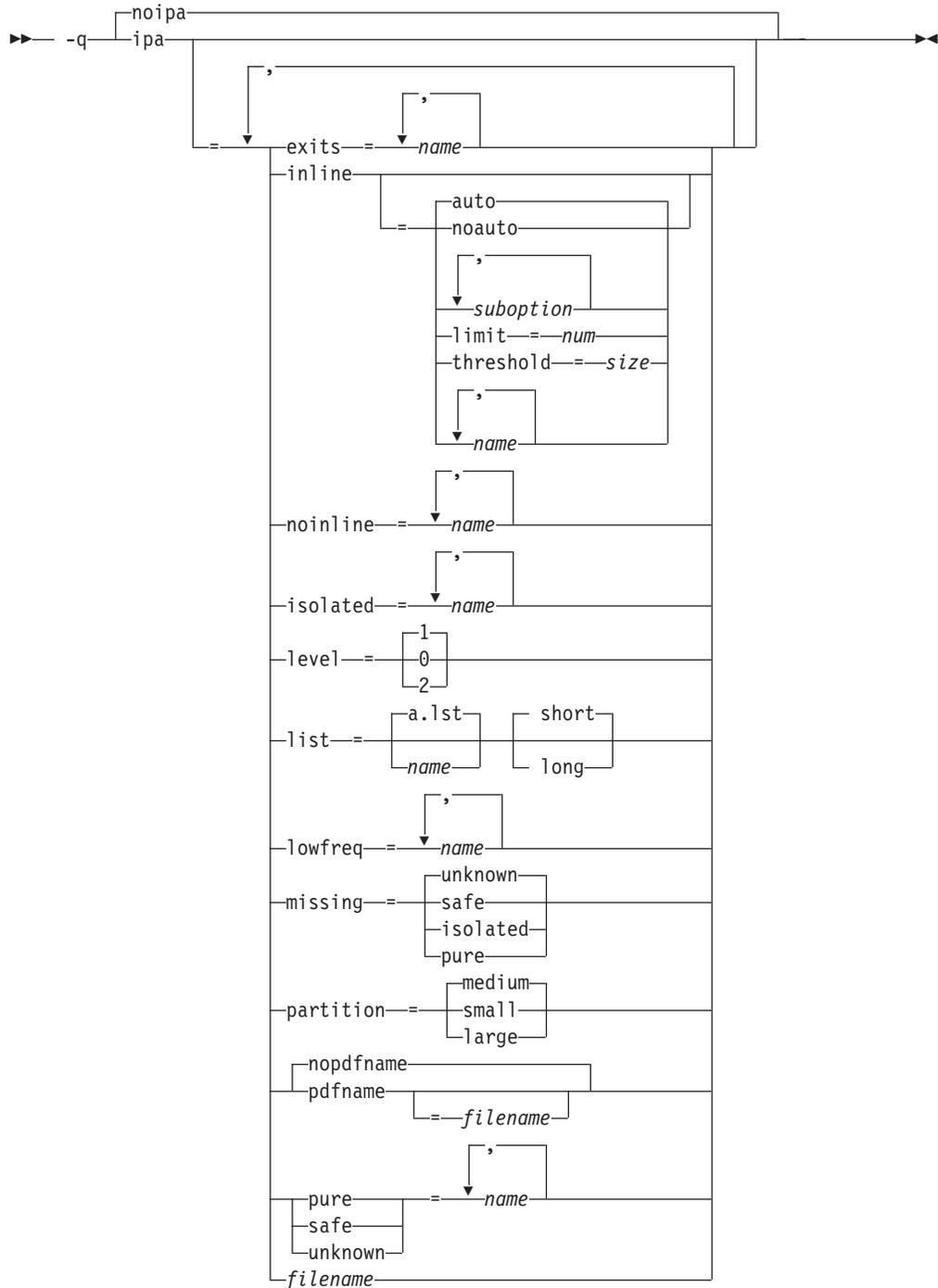
Compile-time syntax



where:

-qipa Compile-time Options	Description
-qipa	Activates interprocedural analysis with the following -qipa <i>suboption</i> defaults: <ul style="list-style-type: none">• 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 <i>suboption</i> defaults: <ul style="list-style-type: none"> • 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= <i>name</i> {, <i>name</i> }	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[= <i>suboption</i>]	Same as specifying the -qinline compiler option, with <i>suboption</i> being any valid -qinline suboption.
inline=limit= <i>num</i>	Changes the size limits that the -Q option uses to determine how much inline expansion to do. This established limit is the size below which the calling procedure must remain. <i>number</i> is the optimizer's approximation of the number of bytes of code that will be generated. Larger values for this number allow the compiler to inline larger subprograms, more subprogram calls, or both. This argument is implemented only when inline=auto is on.
inline=threshold= <i>size</i>	Specifies the upper size limit of functions to be inlined, where <i>size</i> is a value as defined under inline=limit . This argument is implemented only when inline=auto is on.
inline= <i>name</i> {, <i>name</i> }	Specifies a comma-separated list of functions to try to inline, where functions are identified by <i>name</i> .
noinline= <i>name</i> {, <i>name</i> }	Specifies a comma-separated list of functions that must not be inlined, where functions are identified by <i>name</i> .
isolated= <i>name</i> {, <i>name</i> }	Specifies a list of <i>isolated</i> 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: <ul style="list-style-type: none"> • 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= <i>[name]</i> [short long]	<p>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.</p> <p>If listings have been requested (using either the -qlist or -qipa=list options), and <i>name</i> is not specified, the listing file name defaults to a.lst.</p> <p>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.</p>
lowfreq= <i>name</i> {, <i>name</i> }	<p>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.</p>
missing= <i>attribute</i>	<p>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.</p> <p>The following attributes may be used to refine this information:</p> <ul style="list-style-type: none"> • 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 <i>isolated</i>. • pure - Functions which are <i>safe</i> and <i>isolated</i> and which do not indirectly alter storage accessible to visible functions. <i>pure</i> functions also have no observable internal state. • unknown - The default setting. This option greatly restricts the amount of interprocedural optimization for calls to <i>unknown</i> functions. Specifies that the missing functions are not known to be <i>safe</i>, <i>isolated</i>, or <i>pure</i>.
partition=small partition=medium partition=large	<p>Specifies the size of each program partition created by IPA during pass 2.</p>
nopdfname pdfname pdfname= <i>filename</i>	<p>Specifies the name of the profile data file containing the PDF profiling information. If you do not specify <i>filename</i>, the default file name is ._pdf.</p> <p>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.</p>

Link-time Suboptions	Description
<code>pure=name{name}</code>	Specifies a list of <i>pure</i> functions that are not compiled with -qipa . Any function specified as <i>pure</i> must be <i>isolated</i> and <i>safe</i> , and must not alter the internal state nor have side-effects, defined as potentially altering any data visible to the caller.
<code>safe=name{name}</code>	Specifies a list of <i>safe</i> functions that are not compiled with -qipa . Safe functions can modify global variables, but may not call functions compiled with -qipa .
<code>unknown=name{name}</code>	Specifies a list of <i>unknown</i> functions that are not compiled with -qipa . Any function specified as <i>unknown</i> can make calls to other parts of the program compiled with -qipa , and modify global variables and dummy arguments.
<code>filename</code>	<p>Gives the name of a file which contains suboption information in a special format.</p> <p>The file format is the following:</p> <pre># ... 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</pre> <p>where <i>attribute</i> is one of:</p> <ul style="list-style-type: none"> • 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 automatics. 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 its 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 compilers. 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
<i>string</i>	Matches any of the characters specified in <i>string</i> . For example, <code>test</code> will match <code>testimony</code> , <code>latest</code> , and <code>intestine</code> .
<code>^string</code>	Matches the pattern specified by <i>string</i> only if it occurs at the beginning of a line.
<i>string</i> \$	Matches the pattern specified by <i>string</i> only if it occurs at the end of a line.
<i>str.in</i> g	The period (<code>.</code>) matches any single character. For example, <code>t.st</code> will match <code>test</code> , <code>tast</code> , <code>tZst</code> , and <code>t1st</code> .
<i>string</i> \special_char	The backslash (<code>\</code>) can be used to escape special characters. For example, assume that you want to find lines ending with a period. Simply specifying the expression <code>.\$</code> would show all lines that had at least one character of any kind in it. Specifying <code>\.\$</code> escapes the period (<code>.</code>), and treats it as an ordinary character for matching purposes.
[<i>string</i>]	Matches any of the characters specified in <i>string</i> . For example, <code>t[a-g123]st</code> matches <code>tast</code> and <code>test</code> , but not <code>t-st</code> or <code>tAst</code> .

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

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 exists 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 119

“libansi” on page 151

“list” on page 153

“pdf1, pdf2” on page 179

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 263 and “#pragma options” on page 272.

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** keyword **isolated_call** 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.

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 263

“#pragma options” on page 272

See also `__attribute__((const))` and `__attribute__((pure))` in the *Function Attributes* section of the *C/C++ Language Reference*.

keyword

► C ► C++

Purpose

This option controls whether the specified name is treated as a keyword or an identifier whenever it appears in your program source.

Syntax

►► -q $\left\{ \begin{array}{l} \text{keyword} \\ \text{nokeyword} \end{array} \right\} = \text{keyword_name}$ ►►

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
- inline
- 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 *-lkey* option.

Syntax

▶— *-L—directory—* ▶

Notes

If the *-Ldirectory* option is specified both in the configuration file and on the command line, the paths specified in the configuration file are searched first.

Default

The default is to search only the standard directories.

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 133

Appendix C, “Libraries in XL C/C++ Advanced Edition for Mac OS X,” on page 311

► C ► C++

Purpose

Searches the specified library file, `libkey.dylib`, and then `libkey.a` for dynamic linking, or just `libkey.a` for static linking.

Syntax

►► `-lkey` ◀◀

Default

The default is to search only some of the compiler run-time libraries. See the default configuration file for the list of default libraries corresponding to the invocation command being used and the level of the operating system.

Notes

The actual search path can be modified with the `-Ldirectory` .

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`. 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 include my library (`libmylibrary.a`), enter:

```
xlc++ myprogram.C -lmylibrary
```

Related Tasks

“Specify Compiler Options in a Configuration File” on page 17

Related References

“Compiler Command Line Options” on page 35

“B” on page 59

“L” on page 132

Appendix C, “Libraries in XL C/C++ Advanced Edition for Mac OS X,” on page 311

langlvl

► C ► C++

Purpose

Selects the language level for the compilation.

Syntax

►► -q—langlvl=*language*—►►

where values for *language* are described below in the **Notes** section.

See also “#pragma langlvl” on page 265 and “#pragma options” on page 272.

Default

The default language level varies according to the command you use to invoke the compiler:

Invocation	Default language level
xlc++	extended
xlC	extended
xlc	extc89
cc	extended
c89	stdc89
c99	stdc99

Notes

You can also use either of the following preprocessor directives to specify the language level in your source program:

```
#pragma options langlvl=language  
#pragma langlvl(language)
```

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 *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.
saa12	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. Note: The compiler supports all language features specified in the ISO C99 Standard. Note that the Standard also specifies features in the run-time library. These features may not be supported in the current run-time library and operating environment.

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. Note: The compiler supports all language features specified in the ISO C99 Standard. Note that the Standard also specifies features in the run-time library. These features may not be supported in the current run-time library and operating environment.
[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 `\uhhhh` for 16-bit characters, or `\Uhhhhhhhh` 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 *language*:

extended	Compilation is based on strict98 , with some differences to accommodate extended language features.
strict98	Compilation conforms to the ISO C++ standard for C++ programs.

[no]anonstruct

This suboption controls whether anonymous structs and anonymous classes are allowed in your C++ source.

By default, 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]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 suboption can be used to select between old (v3.6 or earlier) and current (v5.0 or later) compiler behaviors.

This suboption is useful for building an application that includes an existing shared library originally built with a v3.6 or earlier version of the IBM C/C++ Compilers. Specifying the **noansisinit** suboption ensures that the behavior of global (including static locals) objects with destructors in your newly-compiled objects are compatible with objects built with earlier compilers.

The default setting is **ansisinit**.

[no]gnu_assert	GNU C portability option to enable or disable support for the following GNU C system identification assertions: <ul style="list-style-type: none">• #assert• #unassert• #cpu• #machine• #system
[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	<p>This suboption enables or disables extern template instantiations.</p> <p>The default setting is gnu_externtemplate when compiling to the extended language level, and nognnu_externtemplate when compiling to the strict98 language .</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>See the following examples:</p> <pre> 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> </pre>
[no]gnu_include_next	GNU C portability option to enable or disable support for the GNU C #include_next preprocessor directive.
[no]gnu_locallabel	GNU C portability option to enable or disable support for locally-declared labels.
[no]gnu_warning	GNU C portability option to enable or disable support for the GNU C #warning preprocessor directive.

[no]oldfriend

This option controls whether friend declarations that name classes without elaborated class names are treated as C++ errors.

By default, 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]oldmath

This suboption controls which versions of math function declarations in `<math.h>` are included when you specify `math.h` as an included or primary source file.

By default, the new standard math functions are used. Build with **-qlanglvl=nooldmath** for strict compliance with the C++ standard.

For compatibility with modules that were built with earlier versions of VisualAge C++ and predecessor products you may need to build with **-qlanglvl=oldmath**.

[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, 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]oldtmpspec

This suboption controls whether template specializations that do not conform to the C++ standard are allowed.

By default, allows these old specializations (**-qlanglvl=nooldtmpspec**). 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=oldtmpspec** 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=nooldtmpspec** 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]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, 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 warning message with the **-qsuppress** option.

Set **anonunion** for compliance with standard C++.

[no]illptom

This suboption controls what expressions can be used to form pointers to members. can accept some forms that are in common use, but do not conform to the C++ standard.

By default, 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++ and 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 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, 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, allows `offsetof` to be used with nonPOD 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, 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]traienum

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=notraienum** 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 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 `\uhhhh` for 16-bit characters, or `\Uhhhhhhh` for 32-bit characters, where *h* represents a hexadecimal digit. Short identifiers of characters are specified by ISO/IEC 10646.

The default setting is **-qlanglvl=noucs**.

[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 #define 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 **-qplusplus** 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

“Compiler Command Line Options” on page 35

“bitfields” on page 60

“chars” on page 68

“digraph” on page 84

“flag” on page 94

“info” on page 114

“inline” on page 119

“M” on page 156

“ro” on page 193

“suppress” on page 210

“#pragma langlvl” on page 265

“#pragma options” on page 272

See also the *IBM C Language Extensions* and *IBM C++ Language Extensions* sections of the *C/C++ Language Reference*.

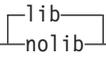
lib

> C > C++

Purpose

Instructs the compiler to use the standard system libraries at link time.

Syntax

►► — -q—  —►►

Notes

If the **-qno lib** 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 81

libansi

► C ► C++

Purpose

Assumes that all functions with the name of an ANSI C library function are in fact the system functions.

Syntax

►► -q no libansi libansi _____►►

See also “#pragma options” on page 272.

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 272

linedebug

C C++

Purpose

Generates line number and source file name information for the debugger.

Syntax

►► -q no linedebug 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 103

“O, optimize” on page 171

“Q” on page 188

“#pragma options” on page 272

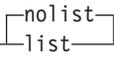
list

► C ► C++

Purpose

Produces a compiler listing that includes an object listing.

Syntax

►► -q  _____ ►►

See also “#pragma options” on page 272.

Notes

When compiling C programs, options that are not defaults appear in all listings, even if **-qno list** is specified.

The **-qno print** compiler option overrides this option.

Example

To compile myprogram.C to produce an object listing enter:

```
xlc++ myprogram.C -qlist
```

Related References

“Compiler Command Line Options” on page 35

“print” on page 185

“#pragma options” on page 272

listopt

> C > C++

Purpose

Produces a compiler listing that displays all options in effect at time of compiler invocation.

Syntax

►► -q no listopt
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 -q listopt
```

Related References

“Compiler Command Line Options” on page 35

“print” on page 185

“Resolving Conflicting Compiler Options” on page 19

longlong

► C ► C++

Purpose

Allows **long long** integer types in your program.

Syntax

►► -q longlong no**longlong** ►►

Default

The default with `xlc`, `xlc++`, `xlC` and `cc` is **-q**longlong****, which defines `_LONG_LONG` (**long long** types will work in programs). The default with `c89` is **-q**no**longlong****** (**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 ints** 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** or **.i** suffix. 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 **-ofile_name** along with **-M**, the **.d** file is placed in the directory implied by **-ofile_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 21

Related References

"Compiler Command Line Options" on page 35

"makedep" on page 161

"o" on page 175

ma

C

Purpose

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

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 53

“`#pragma alloca`” on page 243

macpstr

► C ► C++

Purpose

Converts Pascal string literals into null-terminated strings where the first byte contains the length of the string.

Syntax

► -q nomacpstr
macpstr ►

See also “#pragma options” on page 272.

Notes

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

The final length of the Pascal string literal can be no longer than 255 bytes (the maximum length that can fit in a byte).

For example, the **-qmacpstr** converts:

```
"\pABC"
```

to:

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

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

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

String Literal Processing: The following describes how Pascal string literals are processed.

- Concatenating a Pascal string literal to a normal string gives a non-Pascal string. For example:

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

gives:

```
"ABCpDEF"
```

- A Pascal string literal cannot be concatenated with a **wide** string literal.
- The compiler truncates a Pascal string literal that is longer than 255 bytes (excluding the length byte and the terminating NULL) to 255 characters.
- The compiler ignores the **-qmacpstr** option if **-qmbcs** or **-qdbcs** is used, and issues a warning message.
- Because there is no Pascal-string-literal processing of wide strings, using the escape sequence \p in a wide string literal with the **-qmacpstr** option, generates a warning message and the escape sequence is ignored.
- The Pascal string literal is *not* a basic type different from other C or C++ string literals. After the processing of the Pascal string literal is complete, the resulting

string is treated the same as all other strings. If the program passes a C string to a function that expects a Pascal string, or vice versa, the behavior is undefined.

- Concatenating two Pascal string literals, for example, `strcat()`, does not result in a Pascal string literal. However, as described above, two adjacent Pascal string literals can be concatenated to form one Pascal string literal in which the first byte is the length of the new string literal.
- Modifying any byte of the Pascal string literal after the processing has been completed does not alter the original length value in the first byte.
- No errors or warnings are issued when the bytes of the processed Pascal string literal are modified.
- Entering the characters:

```
'\p' , 'A' , 'B' , 'C' , '\0'
```

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

Example

To compile `mypascal.c` and convert string literals into null-terminated strings, enter:

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

Related References

“Compiler Command Line Options” on page 35

“mbs, dbs” on page 166

“#pragma options” on page 272

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**, or **.i** suffix. 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 **-ofile_name** along with **-qmakedep**, the **.d** file is placed in the directory implied by **-ofile_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: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” on page 21. 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 156

"o" on page 175

"Directory Search Sequence for Include Files Using Relative Path Names" on page 21

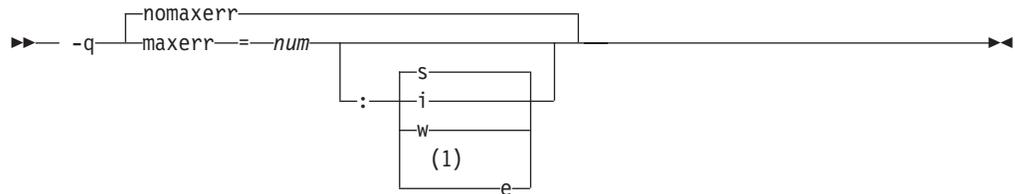
maxerr

C C++

Purpose

Instructs the compiler to halt compilation when *num* errors of a specified severity level or higher is reached.

Syntax



Notes:

- 1 C compilations only

where *num* must be an integer. Choices for severity level can be one of the following:

<i>sev_level</i>	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=5:w -qmaxerr=3 -qhalt=i
```

Related References

“Compiler Command Line Options” on page 35

“flag” on page 94

“halt” on page 107

“Message Severity Levels and Compiler Response” on page 293

maxmem

► C ► C++

Purpose

Limits the amount of memory used for local tables of specific, memory-intensive optimizations to *size* kilobytes. If that memory is insufficient for a particular optimization, the scope of the optimization is reduced.

Syntax

►► -q-maxmem=size 8192►►

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



See also “#pragma options” on page 272.

Notes

Multibyte characters are used in certain languages such as Chinese, Japanese, and Korean.

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 272

mkshrobj

► C ► C++

Purpose

Creates a shared object from generated object files.

Syntax

► `-qmkshrobj` `[=--priority]` ►

where *priority* specifies the priority level for the file. *priority* may be any number from 101 (highest priority-initialized first) to 65535 (lowest priority-initialized last). If no priority is specified the default priority of 65535 is used. The priority is not used when linking shared objects (using the `xlc` command) written in C.

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.

The priority suboption has no effect if you use the `xlc` or `xlc++` command to link with or the shared object has no static initialization.

The `-qpic` and `-qnocommon` compiler options are automatically set when you specify `-qmkshrobj`.

Also, the following related option can be used with the `-qmkshrobj` compiler option:

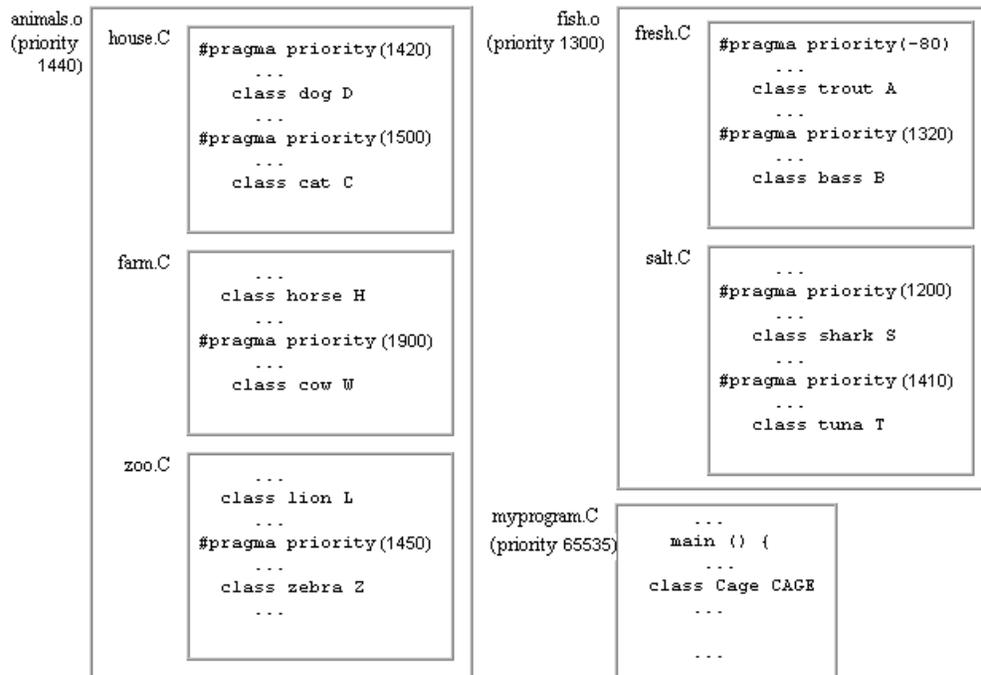
`-oshared_file.o` Is the name of the file that will hold the shared file information. The default is `a.out`.

If you use `-qmkshrobj` to create a shared library, the compiler will call the `libtool` utility with the appropriate options and object files to build a shared object.

Example

The following example shows how to construct a shared library containing two shared objects using the `-qmkshrobj` option, and the `ar` command. The shared library is then linked with a file that contains the main function. Different priorities are used to ensure objects are initialized in the specified order.

The diagram below shows how the objects in this example are arranged in various files.



The first part of this example shows how to use the **-qpriority=N** option and the **#pragma priority(N)** directive to specify the initialization order for objects within the object files.

The example shows how to make two shared objects: `animals.o` containing object files compiled from `house.C`, `farm.C`, and `zoo.C`, and `fish.o` containing object files compiled from `fresh.C` and `salt.C`. The **-qmksbobj=P** option is used to specify the priority of the initialization of the shared objects.

The priority values for the shared objects are chosen so that all the objects in `fish.o` are initialized before the objects in `myprogram.o`, and all the objects in `animals.o` are initialized after the objects in `myprogram.o`.

To specify this initialization order, follow these steps:

1. Develop an initialization order for the objects in `house.C`, `farm.C`, and `zoo.C`:

- a. To ensure that the object `lion L` in `zoo.C` is initialized before any other objects in either of the other two files, compile `zoo.C` using a **-qpriority=N** option with *N* set so both objects have a priority number less than any other objects in `farm.C` and `house.C`:

```
xlc++ zoo.C -c -qpriority=1350
```

- b. Compile the `house.C` and `farm.C` files without specifying the **-qpriority=N** option so objects within the files retain the priority numbers specified by their **#pragma priority(N)** directives:

```
xlc++ house.C farm.C -c
```

- c. Combine these three files in a shared library. Use `xlc++ -qmksbobj` to construct a library `animals.o` with a priority of 1440:

```
xlc++ -qmksbobj=1440 -o animals.o house.o farm.o zoo.o
```

2. Develop an initialization order for the objects in `fresh.C`, and `salt.C`:

a. Compile the fresh.C and salt.C files:

```
xlc++ fresh.C salt.C -c
```

b. To assure that all objects in fresh.C and salt.C are initialized before any other objects, use **xlc++ -qmkshrobj** to construct a library fish.o with a priority of 1300.

```
xlc++ -qmkshrobj=1300 -o fish.o fresh.o salt.o
```

Because the shared library fish.o has a lower priority number (1300) than animals.o (1440), when the files are placed in an archive file with the **ar** command, their objects are initialized first.

3. Compile myprogram.C that contains the function main to produce an object file myprogram.o. By not specifying a priority, this file is compiled with a default priority of 65535, and the objects in main have a priority of 65535.

```
xlc++ myprogram.C -c
```

4. To create a library that contains the two shared objects animals.o and fish.o, you use the **ar** command. To produce an archive file, libzoo.a, enter the command:

```
ar -rv libzoo.a animals.o fish.o
```

where:

- rv The **ar** options. -r replaces a named file if it already appears in the library, and -v writes to standard output a file-by-file description of the making of the new library.
- libzoo.a Is the name you specified for the archive file that will contain the shared object files and their priority levels.
- animals.o Are the two shared files you created with **xlc++ -qmkshrobj**.
- fish.o

5. To produce an executable file, animal_time, so that the objects are initialized in the order you have specified, enter:

```
xlc++ -oanimal_time myprogram.o -L -lzoo
```

6. The order of initialization of the objects is shown in the following table.

Order of Initialization of Objects in libzoo.a			
File	Class Object	Priority Value	Comment
"fish.o"		1300	All objects in "fish.o" are initialized first because they are in a library prepared with -qmkshrobj=1300 (lowest priority number, 1300, specified for any files in this compilation)
	"shark S"	1300(1200)	Initialized first in "fish.o" because within file, #pragma priority(1200)
	"trout A"	1300(1320)	#pragma priority(1320)
	"tuna T"	1300(1410)	#pragma priority(1410)
	"bass B"	1300(1900)	#pragma priority(1900)
"myprog.o"		1400	File generated with priority 1400.
	"CAGE"	1400(1400)	Object generated in main with priority 1400.

Order of Initialization of Objects in libzoo.a			
"animals.o"		1440	File generated with -qmkshrobj=1440
	"lion L"	1440(1350)	Initialized first in file "animals.o" compiled with -qpriority=1350.
	"horse H"	1440(1400)	Follows with priority of .
	"dog D"	1440(1420)	Next priority number (specified by #pragma priority(1420))
	"zebra N"	1440(1450)	Next priority number from #pragma priority(1450)
	"cat C"	1440(1500)	Next priority number from #pragma priority(1500)
	"cow W"	1440()	Next priority number from #pragma priority(65535) (Initialized last)

You can place both nonshared and shared files with different priority levels in the same archive library using the **ar** command.

Related References

"Compiler Command Line Options" on page 35

"common" on page 72

"o" on page 175

"path" on page 178

"pic" on page 184

"priority" on page 186

"#pragma priority" on page 281

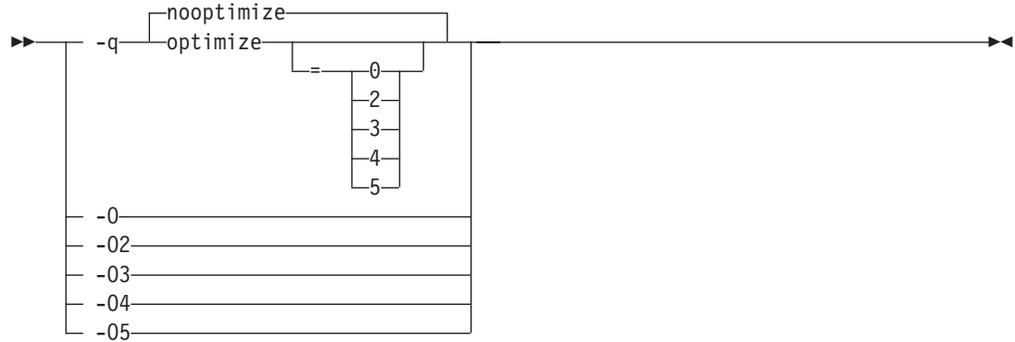
O, optimize

C C++

Purpose

Optimizes code at a choice of levels during compilation.

Syntax



where optimization settings are:

<code>-O</code> <code>-qOPTimize</code>	<p>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.</p> <p>This setting implies <code>-qstrict</code> and <code>-qstrict_induction</code>, unless explicitly negated by <code>-qnostrict_induction</code> or <code>-qnostrict</code>.</p>
<code>-O2</code> <code>-qOPTimize=2</code>	<p>Same as <code>-O</code>.</p>
<code>-O3</code> <code>-qOPTimize=3</code>	<p>Performs additional optimizations that are memory intensive, compile-time intensive, or both. These optimizations are performed in addition to those performed with only the <code>-O</code> option specified. They are recommended when the desire for runtime improvement outweighs the concern for minimizing compilation resources.</p> <p>This is the compiler's highest and most aggressive level of optimization. <code>-O3</code> performs optimizations that have the potential to slightly alter the semantics of your program. It also applies the <code>-O2</code> level of optimization with unbounded time and memory. The compiler guards against these optimizations at <code>-O2</code>.</p> <p>Use the <code>-qstrict</code> option with <code>-O3</code> to turn off the aggressive optimizations that might change the semantics of a program. <code>-qstrict</code> combined with <code>-O3</code> invokes all the optimizations performed at <code>-O2</code> as well as further loop optimizations. The <code>-qstrict</code> compiler option must appear after the <code>-O3</code> option, otherwise it is ignored.</p>

<p>-O3 -qOPTimize=3 (continued)</p>	<p>The aggressive optimizations performed when you specify -O3 are:</p> <ol style="list-style-type: none"> Aggressive code motion, and scheduling on computations that have the potential to raise an exception, are allowed. <p>Loads and floating-point computations fall into this category. This optimization is aggressive because it may place such instructions onto execution paths where they <i>will</i> be executed when they <i>may</i> not have been according to the actual semantics of the program.</p> <p>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 <code>a</code> of 10 elements and load <code>a[60000000003]</code>, which could cause a segmentation violation.</p> <p>The same concepts apply to scheduling.</p> <p>Example:</p> <p>In the following example, at -O2, the computation of <code>b+c</code> is not moved out of the loop for two reasons:</p> <ol style="list-style-type: none"> it is considered dangerous because it is a floating-point operation it does not occur on every path through the loop <p>At -O3, the code is moved.</p> <pre> ... int i ; float a[100], b, c ; for (i = 0 ; i < 100 ; i++) { if (a[i] < a[i+1]) a[i] = b + c ; } ... </pre> Conformance to IEEE rules are relaxed. <p>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.</p> <p>For example, <code>X + 0.0</code> is not folded to <code>X</code> because, under IEEE rules, <code>-0.0 + 0.0 = 0.0</code>, which is <code>-X</code>. In some other cases, some optimizations may perform optimizations that yield a zero result with the wrong sign. For example, <code>X - Y * Z</code> may result in a <code>-0.0</code> where the original computation would produce <code>0.0</code>.</p> <p>In most cases the difference in the results is not important to an application and -O3 allows these optimizations.</p> Floating-point expressions may be rewritten. <p>Computations such as <code>a*b*c</code> may be rewritten as <code>a*c*b</code> 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.</p>
---	---

<p><code>-O3</code>, <code>-qOPTimize=3</code> (continued)</p>	<p>Notes</p> <ul style="list-style-type: none"> • 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.
<p><code>-O4</code> <code>-qOPTimize=4</code></p>	<p>This option is the same as -O3, except that it also:</p> <ul style="list-style-type: none"> • 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 <p>Note: Later settings of -O, -qcache, -qhot, -qipa, -qarch, and -qtune options will override the settings implied by the -O4 option.</p>
<p><code>-O5</code> <code>-qOPTimize=5</code></p>	<p>This option is the same as -O4, except that it:</p> <ul style="list-style-type: none"> • Sets the -qipa=level=2 option to perform full interprocedural data flow and alias analysis. <p>Note: Later settings of -O, -qcache, -qipa, -qarch, and -qtune options will override the settings implied by the -O5 option.</p>
<p><code>-qNOOPTimize</code> <code>-qOPTimize=0</code></p>	<p>Performs only quick local optimizations such as constant folding and elimination of local common subexpressions.</p> <p>This setting implies -qstrict_induction unless -qnostrict_induction is explicitly specified.</p>

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

"Compiler Command Line Options" on page 35

"arch" on page 56

"cache" on page 66

"float" on page 95

"flttrap" on page 98

"g" on page 103

"ignerrno" on page 112

"ignprag" on page 113

"ipa" on page 123

"langlvl" on page 134

"maxmem" on page 165

"strict" on page 208

"strict_induction" on page 209

"tune" on page 223

See also the *Getting Started with Optimization* section in *Getting Started with XL C/C++ Advanced Edition for Mac OS X*



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*. Otherwise, files created by the compiler will take on their default names. 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**.

A *filespec* with a C or C++ source file suffix (**.C**, **.c**, or **.i**), such as *my_text.c* or *bob.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 **-ofilename** 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 executable file produced by the compiler is placed in the **myaccount** directory.

Related References

“Compiler Command Line Options” on page 35

“c” on page 64

“E” on page 87

“P” on page 176

“syntaxonly” on page 212

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* or *file_name.C*.

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, *file_name.ias* input. 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 be 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 87

“o” on page 175

“syntaxonly” on page 212

p

► C ► C++

Purpose

Sets up the object files produced by the compiler for profiling.

Syntax

► — -p ————— ◀

Notes

When compiling and linking in separate steps, the **-p** option must be specified in both steps.

Related References

“Compiler Command Line Options” on page 35

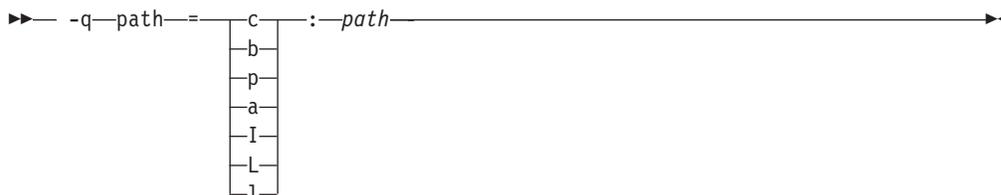
path

> C > C++

Purpose

Constructs alternate program names. The program and directory *path* specified by this option is used in place of the regular program.

Syntax



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

Constructs alternate program names. The program and directory *path* directory are used in place of the regular programs.

The **-qpath** option overrides the **-Fconfig_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 59

“F” on page 93

“t” on page 213

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



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. 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++ Advanced Edition for Mac OS X 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 do 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:

<code>cleanpdf [pathname]</code>	Removes all profiling information from the <i>pathname</i> directory; or if <i>pathname</i> is not specified, from the PDFDIR directory; or if PDFDIR is not set, from the current directory.
	Removing the profiling information reduces the runtime overhead if you change the program and then go through the PDF process again.
	Run this program after compiling with -qpdf2 , or after finishing with the PDF process for a particular application. If you continue using PDF with an application after running cleanpdf , you must recompile all the files with -qpdf1 .
<code>resetpdf [pathname]</code>	Same as <code>cleanpdf [pathname]</code> , described above.

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 123

“O, optimize” on page 171

pg

C C++

Purpose

Sets up the object files for profiling.

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

phsinfo

► C ► C++

Purpose

Reports the time taken in each compilation phase. Phase information is sent to standard output.

Syntax

►► -q nophsinfo
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
```

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

►► -q  ►►

where

- `nopic` Instructs the compiler to not generate Position Independent Code.
- `pic` Instructs the compiler to generate Position Independent Code.

Notes

The `-qp` option is enabled if the `-qmkshrobj` compiler option is specified.

Example

To compile a shared library `libmylib.dylib`, use the following command:

```
xlc mylib.c -qp -Wl, -shared -o libmylib.dylib
```

Refer to the `ld` command in your operating system documentation for more information about the `-shared` option.

Related References

- “Compiler Command Line Options” on page 35
- “mkshrobj” on page 167

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

► -q print 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 58

“list” on page 153

“listopt” on page 154

“source” on page 200

“xref” on page 237

priority

C++

Purpose

Specifies the priority level for the initialization of static constructors

Syntax

► — `-q—priority=—number—` —►

See also “`#pragma priority`” on page 281 and “`#pragma options`” on page 272.

Notes

number Is the initialization priority level assigned to the static constructors 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 272

“`#pragma priority`” on page 281

proto

► c

Purpose

If this option is set, the compiler assumes that all functions are prototyped.

Syntax

► — -q — no 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.

You can obtain 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

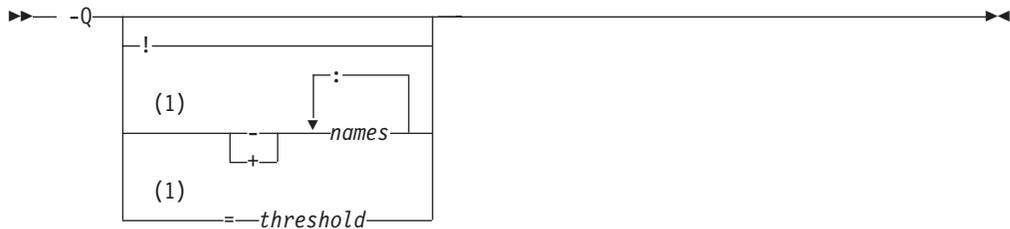
> C > C++

Purpose

In the C language, instructs the compiler to try to inline functions instead of generating calls to a function. Inlining is performed if possible, but, depending on which optimizations are performed, some functions might not be inlined.

In the C++ language, specifies which functions will be inlined instead of generating a call to a function.

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 *function_name*. Separate each *function_name* 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 *function_name* and any other appropriate functions. Each *function_name* 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+function_name** 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 and depends on other options that you select:

- If you specify the **-g** option (to generate debug information), no functions are inlined.
- 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.

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 103

“inline” on page 119

“O, optimize” on page 171

“Q” on page 188

“The `inline`, `_Inline`, `_inline`, and `__inline` Function Specifiers” on page 121

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 optimized. The transformation reports are included as part of the compiler listing.

Syntax

►► -q noreport report _____►►

Notes

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.

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
```

Related References

“Compiler Command Line Options” on page 35

“hot” on page 109

ro

C C++

Purpose

Specifies the storage type for string literals.

Syntax

►► -q ro noro ►►

See also “#pragma options” on page 272.

Default

The default with `xlc`, `xlc++`, `xlC`, and `c89` is `-qro`. The default with `cc` 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 generates 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

“#pragma options” on page 272

roconst

C C++

Purpose

Specifies the storage location for constant values.

Syntax

►► -q roconst
norconst ◀◀

See also “#pragma options” on page 272.

Default

The default with `xl`, `xl++`, `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 193

“#pragma options” on page 272

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

►► -q rtti
nortti ◀◀

where available suboptions are:

nortti	The compiler does not generate the information needed for the RTTI <code>typeid</code> and <code>dynamic_cast</code> operators.
rtti	The compiler generates the information needed for the RTTI <code>typeid</code> and <code>dynamic_cast</code> operators.

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 89

rwvftable

C++

Purpose

If specified, this option instructs the compiler to place virtual function tables into read/write memory.

Syntax

►► -q norwvftable
rwvftable ◀◀

where:

norwvftable	The application does not place virtual function tables into read/write memory.
rwvftable	The application places virtual function tables into read/write memory.

Related References

“Compiler Command Line Options” on page 35

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 103

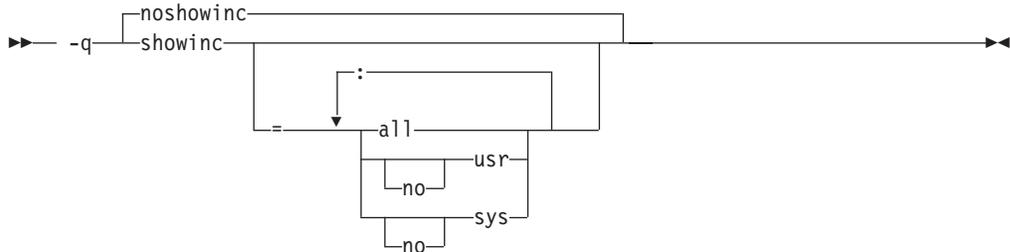
showinc

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 272.

Notes

This options has effect only when the **-qsource** compiler option is 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 200

“#pragma options” on page 272

smallstack

► C ► C++

Purpose

Instructs the compiler to reduce the size of the stack frame.

Syntax

►► -q nosmallstack 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 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 -qsmallstack
```

Related References

“Compiler Command Line Options” on page 35

“g” on page 103

“ipa” on page 123

“O, optimize” on page 171

source

> C > C++

Purpose

Produces a compiler listing and includes source code.

Syntax

►► — -q —  ◀◀

See also “#pragma options” on page 272.

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

The following code causes the parts of the source code between the **#pragma options** 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
```

To compile `myprogram.C` to produce a compiler listing that includes the source for `myprogram.C`, enter:

```
xlc++ myprogram.C -qsource
```

Related References

“Compiler Command Line Options” on page 35

“print” on page 185

“#pragma options” on page 272

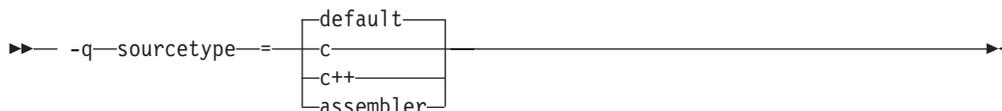
sourcetype

► 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



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 case in 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:

<code>hello.c</code>	The file is compiled as a C file.
<code>hello.C</code>	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 44

spill

C C++

Purpose

Specifies the register allocation spill area as being *size* bytes.

Syntax

►► -q-spill=size ◀◀

See also “#pragma options” on page 272.

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 272

srcmsg

► C

Purpose

Adds the corresponding source code lines to the diagnostic messages in the **stderr** file.

Syntax

►► -q nosrcmsg
srcmsg ◀◀

See also “#pragma options” on page 272.

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 272

staticinline

► C ► C++

Purpose

This option controls whether inline functions are treated as static or extern. By default, XL C/C++ Advanced Edition for Mac OS X treats inline functions as extern.

Syntax

► — -q — 

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

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

►► -q nostatsym
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

stdframework

> C > C++

Purpose

Determines if system default framework directories are searched for header files.

Syntax

►► -q stdframework
nostdframework ◀◀

where:

stdframework	System framework directories are included in the search path when the compiler searches for header files.
nostdframework	System framework directories are not included in the search path when the compiler searches for header files.

Notes

By default, the compiler will search for a header file in the following locations, listed in order of search priority, until it is found:

1. Ordinary header file locations
2. User-defined framework directories (if they exist and are specified by the **-qframeworkdir** compiler option)
3. System-default framework directories, listed in order of priority:
 - a. /Library/Frameworks/
 - b. /Network/Library/Frameworks/
 - c. /System/Library/Frameworks/
4. Subframework directories, if in an umbrella framework

Specifying the **-qnostdframework** compiler option will remove system-default framework directories from the search path.

Related References

“Compiler Command Line Options” on page 35

“framework” on page 100

“frameworkdir” on page 101

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

►► -q stdinc
nostdinc _____►►

See also “#pragma options” on page 272.

Notes

If you specify **-qnostdinc**, the compiler will not search the default search path directories unless you explicitly add them with the **-Idirectory** 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 **-Idirectory** 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 Tasks

“Directory Search Sequence for Include Files Using Relative Path Names” on page 21

Related References

“Compiler Command Line Options” on page 35

“I” on page 110

“idirfirst” on page 111

“#pragma options” on page 272

“Directory Search Sequence for Include Files Using Relative Path Names” on page 21

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 272.

Default

- **-qnostrict** with optimization levels of 3 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 95

“O, optimize” on page 171

“#pragma options” on page 272

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

►► -qnostrict_induction
└──────────┬──────────┘
strict_induction

Default

- **-qnostrict_induction** with optimization levels 3 or higher.
- **-qstrict_induction** otherwise.

Notes

Use of this option is generally not recommended because it can cause considerable performance degradation. If your program is not sensitive to induction variable overflow or wrap-around, you should consider using **-qnostrict_induction** in conjunction with the **-O2** optimization option.

Related References

“Compiler Command Line Options” on page 35

“O, optimize” on page 171

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 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 103

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 87

“o” on page 175

“P” on page 176

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 **-Bprefix** is not specified at all, the prefix of the standard program names is **/lib/n**.

If **-B** is specified but **-tprograms** 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 59

tabsize

► C ► C++

Purpose

Changes the length of tabs as perceived by the compiler.

Syntax

►► -q-tabsize= $\overset{8}{\boxed{n}}$ ◀◀

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

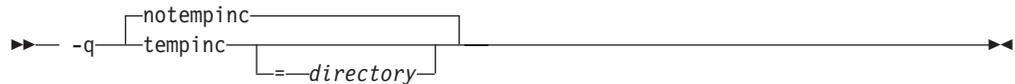
tempinc

C++

Purpose

Generates separate include files for template functions and class declarations, and places these files in a directory which can be optionally specified.

Syntax



Notes

The **-qtempinc** and **-qtemplateregistry** compiler options are mutually exclusive. Specifying **-qtempinc** implies **-qnotemplateregistry**. 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 Tasks

“Structure a Program that Uses Templates” on page 23

“Generate Template Functions Automatically” on page 28

“Define Template Functions Directly in Compilation Units” on page 27

Related References

“Compiler Command Line Options” on page 35

“templateregistry” on page 217

templaterecompile

C++

Purpose

Helps manage dependencies between compilation units that have been compiled using the **-qtemplateregistry** compiler option.

Syntax

► — -q — templaterecompile —
 └──────────────────┘
 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 nominates a single compilation unit to contain the instantiation. No other compilation units will contain this instantiation. Duplication of object code is thereby 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 Tasks

“Structure a Program that Uses Templates” on page 23

“Use the Template Registry to Define Template Functions” on page 26

Related References

“Compiler Command Line Options” on page 35

“templateregistry” on page 217

“tempinc” on page 215

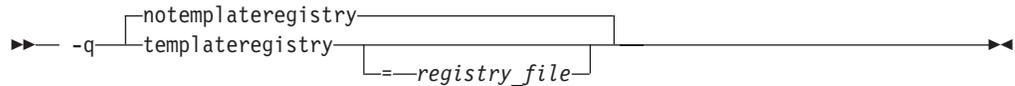
templateregistry

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



Notes

The **-qtempinc** and **-qtemplateregistry** compiler options are mutually exclusive. Specifying **-qtempinc** implies **-qnottemplateregistry**. However, specifying **-qnottempinc** 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.

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 Tasks

“Structure a Program that Uses Templates” on page 23

“Use the Template Registry to Define Template Functions” on page 26

Related References

“Compiler Command Line Options” on page 35

“templaterecompile” on page 216

“tempinc” on page 215

tempmax

C++

Purpose

Specifies the maximum number of template include files to be generated by the `-qtempinc` option for each header file.

Syntax

►► -qtempmax=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 Tasks

“Structure a Program that Uses Templates” on page 23

Related References

“Compiler Command Line Options” on page 35

“tempinc” on page 215

“#pragma implementation” on page 258

threaded

► C ► C++

Purpose

Indicates to the compiler that the program will run in a multi-threaded environment. 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

tmpparse

C++

Purpose

This option controls whether parsing and semantic checking are applied to template definition (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



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 Tasks

“Structure a Program that Uses Templates” on page 23

Related References

“Compiler Command Line Options” on page 35

trigraph

► C ► C++

Purpose

Lets you use trigraph key combinations to represent characters not found on some keyboards.

Syntax

► -q — notrigraph —————►
 └─ trigraph ─┘

Defaults

The default setting for **-qtrigraph** varies according to the command used to invoke the compiler. Defaults are:

- **-qnotrigraph** for `xlcpp`, `xlcpp_r`, `xlC`, `xlC_r`, `xlC`, `xlC_r`, `cc`, and `cc_r`
- **-qtrigraph** for `c89`, `c89_r`, `c99`, and `c99_r`.

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 **-q[lang]vl** compiler option, regardless of where the **-q[no]trigraph** specification appears on the command line.

► C++ The same is true for C++ programs with one exception. When **-q[lang]vl=strict89** appears *after* **-q[no]trigraph** on the command line, the compiler assumes a setting of **-qtrigraph** regardless of the explicit setting of the **-q[no]trigraph** option on the command line.

Examples

1. To disable trigraph character sequences when compiling your program, enter:

```
xlcpp myprogram.C -qnotrigraph
```

2. The following command line invocation results in trigraphs being enabled regardless of the explicit setting of the **-qnotrigraph** compiler option:

```
xlcpp myprogram.C -qnotrigraph -q[lang]vl=strict98
```

Note: This behavior is exclusive to the **strict98** language level. For all other language levels, explicit setting of the **-q[no]trigraph** compiler option overrides the default setting for trigraph support for a given language level. For example, the following invocation causes trigraph support to be disabled:

```
xlc++ myprogram.C -qnotrigraph -qlanglvl=extended
```

3. To disable trigraph character sequences when compiling to the **strict98** language level, specify **-qnotrigraph** on the command line *after* specifying the **strict98** language level. For example, enter:

```
xlc++ myprogram.C -qlanglvl=strict98 -qnotrigraph
```

Related References

“Compiler Command Line Options” on page 35

“digraph” on page 84

“langlvl” on page 134

tune

C C++

Purpose

Specifies the architecture system for which the executable program is optimized.

Syntax



where architecture suboptions are:

auto	Produces object code optimized for the platform on which it is compiled.
g5	Produces object code optimized for Power Mac G5 processors.
ppc970	Generates instructions specific to PowerPC 970 processors.

See also “#pragma options” on page 272.

Default

The default setting of the **-qtune** option depends on the setting of the **-qarch** option.

- If **-qtune** is specified without **-qarch**, the compiler uses **-qarch=ppcv**.
- If **-qarch** is specified without **-qtune**, the compiler uses the default tuning option for the specified architecture. Listings will show only: TUNE=DEFAULT

Default **-qtune** settings for specific **-qarch** settings are described in “Acceptable Compiler Mode and Processor Architecture Combinations” on page 290.

Notes

You can use **-qtune=suboption** with **-qarch=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.

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 Tasks

“Specify Compiler Options for Architecture-Specific Compilation” on page 19

Related References

“Compiler Command Line Options” on page 35

“arch” on page 56

“Acceptable Compiler Mode and Processor Architecture Combinations” on page 290

U

C C++

Purpose

Undefines the identifier *name* defined by the compiler or by the **-Dname** option.

Syntax

► — **-U**—*name* —►

Notes

The **-Uname** 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 **-Dname** option.

The identifier name can also be undefined in your source program using the **#undef** preprocessor directive.

The **-Uname** option has a higher precedence than the **-Dname** option.

Example

To compile myprogram.c so that the definition of the name **COUNT**, is nullified, enter:

```
xlc myprogram.c -UCOUNT
```

For example if the option **-DCOUNT=1000** is used, a source line **#undefine COUNT** is generated at the top of the source.

Related References

“Compiler Command Line Options” on page 35

“D” on page 82

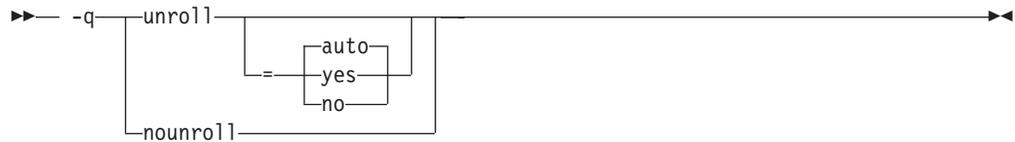
unroll

C C++

Purpose

Unrolls inner loops in the program. This can help improve program performance.

Syntax



where:

<code>-qunroll=auto</code>	Leaves the decision to unroll loops to the compiler. This is the default if the <code>-qunroll</code> compiler option is not specified.
<code>-qunroll</code> or <code>-qunroll=yes</code>	Is a suggestion to the compiler to unroll loops.
<code>-qnounroll</code> or <code>-qunroll=no</code>	Instructs the compiler to not unroll loops.

See also “#pragma unroll” on page 288 and “#pragma options” on page 272.

Notes

Specifying `-qunroll` 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 288 for examples of how program code is unrolled by the compiler.

Related References

"Compiler Command Line Options" on page 35

"#pragma options" on page 272

"#pragma unroll" on page 288

unwind

► C ► C++

Purpose

Informs the compiler that the stack can be unwound while a routine in the compilation is active.

Syntax

►► -q unwind
noundwind ◀◀

Notes

The default is **-qunwind**.

Specifying **-qnoundwind** can improve optimization of non-volatile register saves and restores.

► C++ For C++ programs, specifying **-qnoundwind** also implies **-qnoeh**.

Related References

“Compiler Command Line Options” on page 35

“eh” on page 89

upconv

► C

Purpose

Preserves the **unsigned** specification when performing integral promotions.

Syntax

►► -q noupconv
upconv ◀◀

See also “#pragma options” on page 272.

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=ext**, 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 bitfield** 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 134

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 45

"v" on page 230

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 45

“V” on page 229

vftable

► C++

Purpose

Controls the generation of virtual function tables.

Syntax

► -q { novftable | vftable } ◀

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

►► -q vrsave
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 244

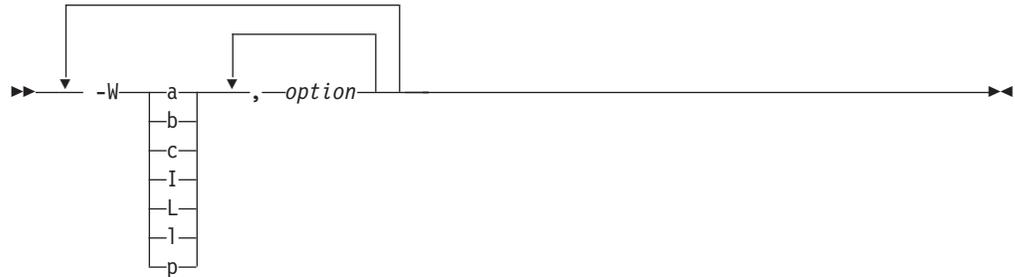
W

C C++

Purpose

Passes the listed option to a designated compiler program.

Syntax



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 —▶

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 94

warnfourcharconst

► C ► C++

Purpose

Enables or disables diagnostic messages issued when the compiler finds four-character constants in the program source.

Syntax

►► -q nowarnfourcharconst warnfourcharconst ►►

Notes

A character constant is represented by enclosing a *single* character within a pair of apostrophes. Defining a constant with more than one character enclosed within a pair of apostrophes ordinarily causes the compiler to issue a warning.

Mac OS X framework headers frequently define four-character constants, which are constants defined by enclosing exactly *four* characters within a pair of apostrophes. Compiling with **-qnowarnfourcharconst** instructs the compiler to suppress warning messages when four-character constants are encountered in program source code.

Example

In the following code segment:

```
enum test { a = 'a',  
            b = 'ab',  
            c = 'abc',  
            d = 'abcd',  
            e = 'abcde'  
};  
  
int main() {  
    return 0;  
}
```

- Compiling with **-qnowarnfourcharconst** in effect results in warnings being issued for variables b, c, and e.
- Compiling with **-qwarnfourcharconst** in effect results in warnings being issued for variables b, c, d, and e.

Related References

“Compiler Command Line Options” on page 35

xcall

> C > C++

Purpose

Generates code to static routines within a compilation unit as if they were external routines.

Syntax

►► -q noxcall xcall _____ ◄◄

Notes

-qxcall generates slower code than -qnoxcall.

Example

To compile myprogram.c so 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 272.

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:

Identifier name	Description of the item
xy	auto int in function adder 0-59Y 0-36.12Z 0-48.12Z

Function invocation
Column number
Line number
File
Function definition

Related References

- “Compiler Command Line Options” on page 35
- “print” on page 185
- “#pragma mc_func” on page 270
- “#pragma options” on page 272

y

> C > C++

Purpose

Specifies the compile-time rounding mode of constant floating-point expressions.

Syntax



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

Z

► C ► C++

Purpose

This option specifies a prefix for the library search path.

Syntax

►— *-Z—string* —►

Notes

This option is useful when developing a new version of a library. Usually you use it to build on one level of an operating system but run on a different level, so that you can search a different path on the development platform than on the target platform. This is possible because the prefix is not stored in the executable.

If you use this option more than once, the strings are appended to each other in the order specified and then they are added to the beginning of the library search paths.

Related References

“Compiler Command Line Options” on page 35

Appendix C, “Libraries in XL C/C++ Advanced Edition for Mac OS X,” on page 311

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
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#pragma align	Aligns data items within structures.
<div style="display: flex; justify-content: space-between;"> > C </div>	#pragma alloca	Provides an inline version of the function <code>alloca(size_t size)</code> .
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#pragma altivec_vrsave	This pragma is accepted and ignored.
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#pragma chars	Sets the sign type of character data.
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#pragma comment	Places a comment into the object file.
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#pragma complexgcc	Instructs the compiler how to pass parameters when calling complex math functions.
<div style="display: flex; justify-content: space-between;"> > C++ </div>	#pragma define	Forces the definition of a template class without actually defining an object of the class.
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#pragma disjoint	Lists the identifiers that are not aliased to each other within the scope of their use.
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#pragma enum	Specifies the size of enum variables that follow.
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#pragma execution_frequency	Lets you mark program source code that you expect will be either very frequently or very infrequently executed.
<div style="display: flex; justify-content: space-between;"> > C++ </div>	#pragma hashome	Informs the compiler that the specified class has a home module that will be specified by the IsHome pragma.
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#pragma ibm snapshot	Sets a debugging breakpoint at the point of the pragma, and defines a list of variables to examine when program execution reaches that point.
<div style="display: flex; justify-content: space-between;"> > C++ </div>	#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.
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#pragma info	Controls the diagnostic messages generated by the info(...) compiler options.
<div style="display: flex; justify-content: space-between;"> > C++ </div>	#pragma ishome	Informs the compiler that the specified class's home module is the current compilation unit.
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#pragma isolated_call	Lists functions that do not alter data objects visible at the time of the function call.
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#pragma langlvl	Selects the C or C++ language level for compilation.
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#pragma leaves	Takes a function name and specifies that the function never returns to the instruction after the function call.
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#pragma map	Tells the compiler that all references to an identifier are to be converted to a new name.
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#pragma mc_func	Lets you define a function containing a short sequence of machine instructions.

Language Application	#pragma	Description
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#pragma options	Specifies options to the compiler in your source program.
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#pragma option_override	Specifies alternate optimization options for specific functions.
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#pragma pack	Modifies the current alignment rule for members of structures that follow this pragma.
<div style="display: flex; justify-content: space-between;"> > C++ </div>	#pragma priority	Specifies the order in which static objects are to be initialized at run time.
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#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.
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#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 .
<div style="display: flex; justify-content: space-between;"> > C++ </div>	#pragma report	Controls the generation of specific messages.
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#pragma strings	Sets storage type for strings.
<div style="display: flex; justify-content: space-between;"> > C > C++ </div>	#pragma unroll	Unrolls inner loops in the program. This can help improve program performance.

Related Tasks

“Specify Compiler Options in Your Program Source Files” on page 17

#pragma align

C C++

Description

The **#pragma align** directive specifies how the compiler should align data items within structures.

Syntax

```
▶▶ #pragma align ( ( power | natural | mac68k | bit_packed | reset ) ) ▶▶
```

See also “#pragma options” on page 272.

Notes

The default setting for **#pragma align** is **power**.

Within your source file, you can 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. 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.

For more information, see “align” on page 49.

Related References

“General Purpose Pragmas” on page 240

“align” on page 49

See also:

- The *Data Mapping and Storage* section of the *Programming Tasks* manual.
- `__attribute__((aligned))` in the *Variable Attributes* section of the *C/C++ Language Reference*.
- `__attribute__((packed))` in the *Variable Attributes* section of the *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 either of the `-qalloca` or `-ma` compiler options to have the compiler provide an inline version of `alloca`.

Once specified, it 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 240

“`alloca`” on page 53

“`ma`” on page 158

#pragma altivec_vrsave

C C++

Description

When the `#pragma altivec_vrsave` directive is enabled, function prologs and epilogs include code to maintain the VRSAVE register.

Syntax

► #pragma altivec_vrsave { on | 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 240

“altivec” on page 54

“vrsave” on page 232

#pragma chars

► C ► C++

Description

The **#pragma chars** directive sets the sign type of char objects to be either **signed** or **unsigned**.

Syntax

```
► #pragma chars (signed | unsigned) ◀
```

Notes

This pragma must appear before any source statements, in order for this pragma to take effect

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 240

“chars” on page 68

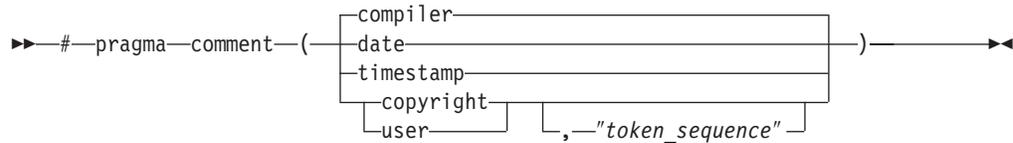
#pragma comment

> C > C++

Description

The **#pragma comment** directive places a comment into the target or object file.

Syntax



where suboptions do the following:

- | | |
|------------------------|---|
| <code>compiler</code> | The name and version of the compiler is appended to the end of the generated object module. |
| <code>date</code> | The date and time of compilation is appended to the end of the generated object module. |
| <code>timestamp</code> | The date and time of the last modification of the source is appended to the end of the generated object module. |
| <code>copyright</code> | The text specified by the <i>token_sequence</i> is placed by the compiler into the generated object module and is loaded into memory when the program is run. |
| <code>user</code> | The text specified by the <i>token_sequence</i> is placed by the compiler into the generated object but is <i>not</i> loaded into memory when the program is run. |

Related References

"General Purpose Pragmas" on page 240

#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 a stack. This instructs the compiler to use GCC conventions when passing and returning complex parameters.
- off Pushes **-qfloat=nocomplexgcc** onto a 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 Pragas” on page 240

“complexgccincl” on page 75

“float” on page 95

#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 global 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 Pragas” on page 240

#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 240

“ignprag” on page 113

“alias” on page 47

#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 ( int | 1 | 2 | 4 | 8 | int long | pop | reset ) ▶▶
```

where available suboptions are:

<small>small</small>	enum size is the smallest integral type that can contain all variables.
<small>int</small>	enum size is 4
<small>1</small>	enum size is 1
<small>2</small>	enum size is 2
<small>4</small>	enum size is 4
<small>8</small>	enum size is 8.
<small>int</small>	enum size is 4
<small>int long</small>	Specifies that enumerations occupy 8 bytes of storage and are represented by long long if the range of the constants exceed the limit for int . Otherwise, the enumerations occupy 4 bytes of storage and are represented by int .
<small>small</small>	enum size is the smallest integral type that can contain all variables.
<small>pop</small>	the option will reset the enum size to the one before the previously set enum size.
<small>reset</small>	the option is an alternative method of resetting the enum size to the one before the previously set enum size. 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 **#includes** it.

The **#pragma options enum=** directive can be used instead of **#pragma enum**. The two pragmas are interchangeable.

The following are invalid enumerations or invalid 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
```

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 default */
```

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 `test_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.

Related References

"General Purpose Pragmas" on page 240

"enum" on page 90

"#pragma options" on page 272

#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 branching to Block A or Block B in the decision "if (**boo**)". The second pragma is ignored by the compiler.

Related References

"General Purpose Pragmas" on page 240

#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**.

Related References

"General Purpose Pragmas" on page 240

"#pragma ishome" on page 262

#pragma ibm snapshot

► C ► C++

Description

The **#pragma ibm snapshot** directive sets a debugging breakpoint at the point of the pragma, and defines a list of variables to examine when program execution reaches that point.

Syntax

```
►► #pragma ibm snapshot ( variable_name )
```

where *variable_name* is a predefined or namespace scope type. Class, structure, or union members cannot be specified.

Notes

Variables specified in **#pragma ibm snapshot** can be observed in the debugger, but should not be modified. Modifying these variables in the debugger may result in unpredictable behavior.

Example

```
#pragma ibm snapshot(a, b, c)
```

Related References

“General Purpose Pragmas” on page 240

#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 240

“tempmax” on page 218

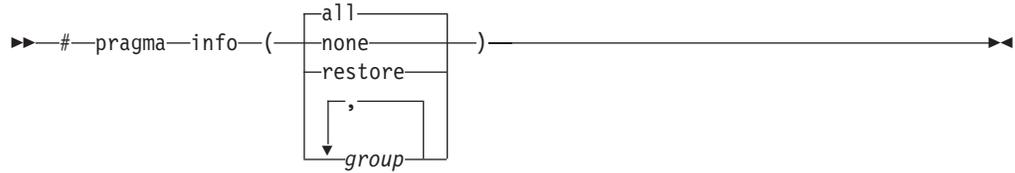
#pragma info

C C++

Description

The **#pragma info** directive instructs the compiler to produce or suppress specific groups of compiler messages.

Syntax



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.

<i>group</i>	Type of messages returned or suppressed
c99 noc99	C code that may behave differently between C89 and C99 language levels.
cls nocls	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 noui	Uninitialized variables
use nouse	Unused auto and static variables
vft novft	Generation of virtual function tables

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

“General Purpose Pragmas” on page 240

“info” on page 114

#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. The advantage of this is the minimization of code.

Syntax

▶▶ #pragma ishome (*className*) ▶▶

where:

className 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**.

Related References

"General Purpose Pragmas" on page 240

"#pragma hashome" on page 256

#pragma isolated_call

C C++

Description

The **#pragma isolated_call** directive lists 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 240

“ignprag” on page 113

“isolated_call” on page 130

See also `__attribute__((const))` and `__attribute__((pure))` in the *Function Attributes* section of the *C/C++ Language Reference*.

#pragma langlvl

► C ► C++

Description

The `#pragma langlvl` directive selects the C or 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*:

<code>classic</code>	Allows the compilation of non-stdc89 programs, and conforms closely to the K&R level preprocessor.
<code>extended</code>	Provides compatibility with the RT compiler and classic . This language level is based on C89.
<code>saa</code>	Compilation conforms to the current SAA C CPI language definition. This is currently SAA C Level 2.
<code>saa12</code>	Compilation conforms to the SAA C Level 2 CPI language definition, with some exceptions.
<code>stdc89</code>	Compilation conforms to the ANSI C89 standard, also known as ISO C90.
<code>stdc99</code>	Compilation conforms to the ISO C99 standard. Note: The compiler supports all language features specified in the ISO C99 Standard. Note that the Standard also specifies features in the run-time library. These features may not be supported in the current run-time library and operating environment.
<code>extc89</code>	Compilation conforms to the ANSI C89 standard, and accepts implementation-specific language extensions.
<code>extc99</code>	Compilation conforms to the ISO C99 standard, and accepts implementation-specific language extensions. Note: The compiler supports all language features specified in the ISO C99 Standard. Note that the Standard also specifies features in the run-time library. These features may not be supported in the current run-time library and operating environment.

► **C++** For C++ programs, you can specify one of the following values for *language*:

<code>extended</code>	Compilation is based on strict98 , with some differences to accommodate extended language features.
<code>strict98</code>	Compilation conforms to the ISO C++ standard for C++ programs.

Default

The default language level varies according to the command you use to invoke the compiler:

Invocation	Default language level
<code>xlC++</code> , <code>xlC</code>	<code>extended</code>
<code>xlC</code>	<code>extc89</code>
<code>cc</code>	<code>extended</code>

<code>c89</code>	<code>stdc89</code>
<code>c99</code>	<code>stdc99</code>

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 Pragma” on page 240

“E” on page 87

“langlvl” on page 134

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 similarly 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 240

See also `__attribute__((noreturn))` in the *Function Attributes* section of the *C/C++ Language Reference*.

#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:

<i>identifier</i>	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.
<i>function_signature</i>	A name of a function or operator with internal linkage. The name can be qualified.
<i>name</i>	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 3, 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 240

#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:

<i>function</i>	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.
<i>instruction_seq</i>	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 some 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 240

“#pragma isolated_call” on page 263

“#pragma reg_killed_by” on page 283

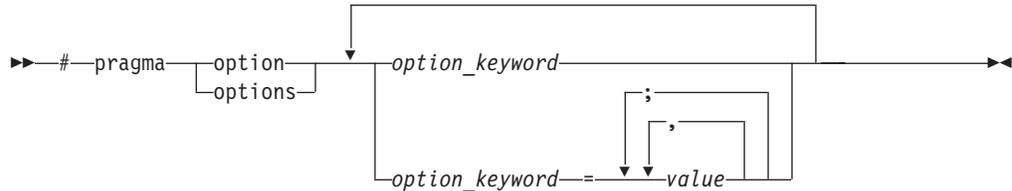
#pragma options

C C++

Description

The **#pragma options** directive specifies compiler options for your source program.

Syntax



Notes

By default, pragma options generally apply to the entire source program. Some pragmas must be specified before any program source statements. For the documentation for more information on pragma options.

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 program source code. 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 <i>option_keyword</i>	Compiler option equivalent	Description
C	C++	align= <i>option</i>	-qalign	Specifies what aggregate alignment rules the compiler uses for file compilation.

Language Application		Valid settings for #pragma options <i>option_keyword</i>	Compiler option equivalent	Description
► C	► C++	[no]ansialias	-qalias	Specifies whether type-based aliasing is to be used during optimization.
► C		assert= <i>option</i>	-qalias	Instructs 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= <i>option</i>	-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= <i>option</i>	-qenum See also #pragma enum	Specifies the amount of storage occupied by the enumerations.

Language Application		Valid settings for #pragma options <i>option_keyword</i>	Compiler option equivalent	Description
► C	► C++	flag= <i>option</i>	-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] <i>option</i>	-qfloat	Specifies various floating point options to speed up or improve the accuracy of floating point operations.
► C	► C++	[no]flttrap= <i>option</i>	-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.
► 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++	[no]ignprag	-qignprag	Instructs the compiler to ignore certain pragma statements.
► C	► C++	[no]info= <i>option</i>	-qinfo See also #pragma info	Produces informational messages.
► C	► C++	initauto= <i>value</i>	-qinitauto	Initializes automatic storage to a specified hexadecimal byte value.
► C	► C++	isolated_call= <i>names</i>	-qisolated_call See also #pragma isolated_call	Specifies functions in the source file that have no side effects.

Language Application		Valid settings for #pragma options <i>option_keyword</i>	Compiler option equivalent	Description
► C	► 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]macpstr	-qmacpstr	Converts Pascal string literals into null-terminated strings where the first byte contains the length of the string.
► C	► C++	[no]maxmem= <i>number</i>	-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 <i>option_keyword</i>	Compiler option equivalent	Description
► C	► C++	[no]optimize optimize= <i>number</i>	-O, optimize	Specifies the optimization level to apply to a section of program code. The compiler will accept the following values for <i>number</i> : <ul style="list-style-type: none"> • 0 - sets level 0 optimization • 2 - sets level 2 optimization • 3 - sets level 3 optimization If no value is specified for <i>number</i> , the compiler assumes level 2 optimization.
	► C++	priority= <i>number</i>	-qpriority See also “#pragma priority” on page 281	Specifies the priority level for the initialization of static constructors
► 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= <i>number</i>	-qspill	Specifies the size of the register allocation spill area.
► C		[no]srcmsg	-qsrcmsg	Adds the corresponding source code lines to the diagnostic messages in the stderr file.
► 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.

Language Application		Valid settings for #pragma options <i>option_keyword</i>	Compiler option equivalent	Description
► C	► C++	tune= <i>option</i>	-qtune	Specifies the architecture for which the executable program is optimized.
► C	► C++	[no]unroll unroll= <i>number</i>	-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.
► 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 240

“E” on page 87

#pragma option_override

C C++

Description

The `#pragma option_override` directive lets you specify alternate optimization options for specific functions.

Syntax

```
▶▶ #pragma option_override (—func_name—[, “—option—”]—)▶▶
```

Notes

By default, optimization options specified on the command line apply to the entire source program. This option lets you override those default settings for specified functions (*func_name*) in your program.

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 or greater 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.

The function specified in this pragma can not be overloaded. Member functions are not supported.

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 Pragas” on page 240

“O, optimize” on page 171

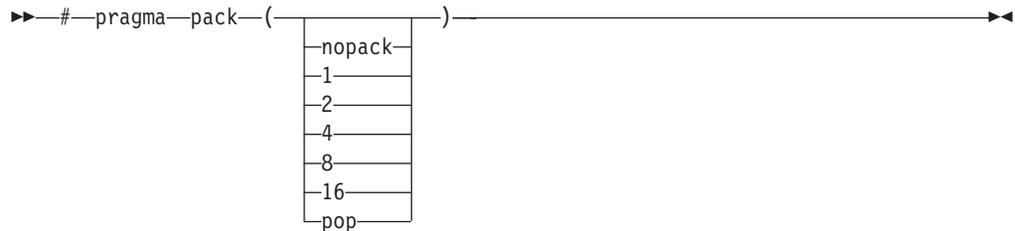
#pragma pack

C C++

Description

The **#pragma pack** directive modifies the current alignment rule for members of structures following the directive.

Syntax



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.
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 () has the same effect as #pragma (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 bitfields to cross bitfield container boundaries.

Examples

1. In the code shown below, the structure S2 will have its members packed to 1-byte, but structure S1 will not be affected. This is because the declaration for S1 began before the pragma directive. However, since the declaration for S2 began after the pragma directive, it is affected.

```
struct s_t1 {
    char a;
    int b;
    #pragma pack(1)
    struct s_t2 {
        char x;
        int y;
    } S2;
    char c;
    int b;
} 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:

```
sizeof S = 16
offsetof a = 0
offsetof b = 4
offsetof c = 8
offsetof d = 12
align of a = 1
align of b = 4
align of c = 2
align of d = 4
```

With #pragma pack(1):

```
sizeof S = 11
offsetof a = 0
offsetof b = 1
offsetof c = 5
offsetof d = 7
align of a = 1
align of b = 1
align of c = 1
align of d = 1
```

Related References

“General Purpose Pragmas” on page 240

“align” on page 49

“#pragma options” on page 272

#pragma priority

C++

Description

The **#pragma priority** directive specifies the order in which static objects are to be initialized at run time.

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 specified applies to all runtime static initialization in the current compilation unit.

Any global object declared before another object in a file is constructed first. Use **#pragma priority** to specify the construction order of objects within a single compilation unit. However, if the user is creating an executable or shared library target from source files, the compiler will check dependency ordering, which may override **#pragma priority**.

For example, if the constructor to object B is passed object A as a parameter, 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.

To ensure that the objects are always constructed from top to bottom in a file, the compiler enforces the restriction that the priority specifies all objects before and all objects after it until the next **#pragma** (that is encountered) is at that priority.

init_priority() Attribute: You can also use the **init_priority()** attribute to assign a priority level to shared variables with class type. Priority levels may range from 101 to 65535. For example:

```
A a_attribute__((init_priority(500)));
```

If no priority level is assigned with either the **init_priority()** attribute or **#pragma priority**, a default priority of 65535 is assumed.

Using the **init_priority()** attribute can cause initialization to occur outside of declaration order, which violates the C++ standard. You can use the compiler messages produced by the **-qinfo=por** compiler option to identify initializations occurring out of order.

Example

```
#pragma priority(1001)
```

Related References

“General Purpose Pragmas” on page 240

“info” on page 114

#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 `setjmp`.

Syntax

▶▶ #pragma reachable (*function*) ▶▶

Related References

“General Purpose Pragmas” on page 240

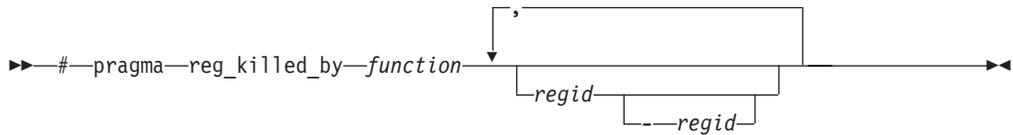
#pragma reg_killed_by

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) <ul style="list-style-type: none">• 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 <code>#pragma reg_killed_by</code>.
fp[0-31]	Floating point registers (FPR) <ul style="list-style-type: none">• Of the 32 machine registers, only fp0-fp13 can be specified by <code>#pragma reg_killed_by</code>.
fs	Floating point status and control register (FPSCR)
lr	Link register (LR)
gr[0-31]	General purpose registers (GPR) <ul style="list-style-type: none">• Of the 32 machine registers, only gr0 and gr3-gr12 can be specified by <code>#pragma reg_killed_by</code>.
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 Pragas” on page 240

“#pragma mc_func” on page 270

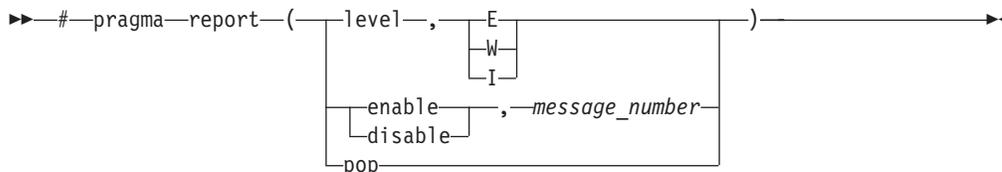
#pragma report

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



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) 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(E) 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(E) 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. **#pragma info** declares all messages to be information messages. The pragma report instructs the compiler to to display only those messages with a severity of 'W' or warning messages. In this case, none of the messages will be displayed.

```
1 #pragma info(all)
2 #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.

```
1 #pragma report(enable, CPPC1000) // enables message number CPPC1000
2 #pragma report(level, E) // display only error messages.
```

Changing the order of the code like so:

```
1 #pragma report(level, E)
2 #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 classified as. In line 3, a new report level of 'E' is set, indicating that 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 240

"flag" on page 94

#pragma strings

► C ► C++

Description

The **#pragma strings** directive sets storage type for strings. It specifies that the compiler can place strings into read-only memory or must place strings into read/write memory.

Syntax

```
►► #pragma strings ( [writeable] | [readonly] )
```

Notes

Strings are read-only by default.

This pragma must appear before any source statements in order to have effect.

Example

```
#pragma strings(writeable)
```

Related References

“General Purpose Pragmas” on page 240

#pragma unroll

C C++

Description

The **#pragma unroll** directive is used to unroll inner loops in your program, which can help improve program performance.

Syntax

```
▶ #pragma [nounroll | unroll(n)]
```

where *n* is the loop unrolling factor. The value of *n* is a positive scalar integer or compile-time constant initialization expression. If *n* is not specified, and if optimization is set at **-O3** or higher, the optimizer determines an appropriate unrolling factor for each 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.

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

“General Purpose Pragmas” on page 240

“unroll” on page 225

Acceptable Compiler Mode and Processor Architecture Combinations

You can use the **-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:

- **-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 machines share a common set of instructions, but may also include additional instructions unique to a given processor or processor family.

If you want to generate code optimized specifically for a particular processor, acceptable combinations of **-qarch**, and **-qtune** compiler options are shown in the following table.

Related Tasks

“Specify Compiler Options for Architecture-Specific Compilation” on page 19

Related References

“Compiler Command Line Options” on page 35

“arch” on page 56

“tune” on page 223

Acceptable -qarch/-qtune Combinations				
-qarch option		Predefined Macro(s)	Default -qtune setting	Available -qtune setting(s)
ppcv		_ARCH_PPCV	ppc970	auto ppc970 g5
	g5	_ARCH_PPCV _ARCH_G5	ppc970	auto ppc970 g5
	ppc970	_ARCH_PPCV _ARCH_G5 _ARCH_PPC970	ppc970	auto ppc970 g5

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 294
- “Compiler Message Format” on page 294

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 valid, but possibly unintended, conditions.
E	Error	 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	The compiler halts. An internal compiler error has occurred. <ul style="list-style-type: none">• 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 294

“Compiler Message Format” on page 294

“halt” on page 107

“maxerr” on page 163

“haltonmsg” on page 108

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 293

"Compiler Message Format"

"halt" on page 107

"maxerr" on page 163

"haltonmsg" on page 108

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

<i>dd</i>	is a two-digit code indicating the XL C/C++ Advanced Edition for Mac OS X component that issued the message. <i>dd</i> 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).
<i>nnn</i>	is the message number
<i>severity</i>	is a letter representing the severity of the error
<i>text</i>	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 293

"Compiler Return Codes" on page 294

"halt" on page 107

"maxerr" on page 163

"haltonmsg" on page 108

Part 4. Appendixes

Appendix A. Predefined Macros

Predefined macros fall into two major categories: those related to the platform and those related to language features. Platform-specific macros are described here.

For information about language-specific macros, see the *Preprocessor Directives* section of the *C/C++ Language Reference*.

Macros related to the platform

The following predefined macros are provided to facilitate porting applications developed with GNU C.

Predefined Macro Name	Description
<code>__APPLE__</code>	Defined to 1 on the Mac OS X platform.
<code>__ALTIVEC__</code>	Defined to 1 by the compiler if AltiVec support is enabled.
<code>__BASE_FILE__</code>	Defined to the fully qualified filename of the primary source file.
<code>_BIG_ENDIAN</code>	Defined to 1.
<code>__BIG_ENDIAN__</code>	Defined to 1.
<code>_BSD_WCHAR_T_DEFINED_</code>	Defined to 1 by the compiler. This macro is defined in the configuration file, and is used by <code>/usr/include/stddef.h</code> and <code>/usr/include/stdlib.h</code> to avoid the definition of <code>wchar_t</code> .
<code>__CHAR_UNSIGNED__</code>	 Defined to 1 if <code>-qchars=unsigned</code> or <code>#pragma chars(unsigned)</code> is in effect. This macro is undefined if <code>-qchars=signed</code> or <code>#pragma chars(signed)</code> is in effect.
<code>__EXCEPTIONS</code>	Defined to 1. This macro is set if the <code>-qeh</code> option is in effect.
<code>__GNU_SOURCE</code>	Defined to 1 on this platform. This macro must be defined because the definitions of other macros are contingent on it.
<code>__GNUC__</code>	Defined to 3 while the compiler is processing header files that are in a directory specified by <code>-qgcc_c_stdinc</code> or <code>-qgcc_cpp_stdinc</code> . Otherwise it is not defined.
<code>__GNUC_MINOR__</code>	Defined to 3 while the compiler is processing header files that are in a directory specified by <code>-qgcc_c_stdinc</code> or <code>-qgcc_cpp_stdinc</code> . Otherwise it is not defined.
<code>__GNUG__</code>	Defined to 3 while the compiler is processing header files that are in a directory specified by <code>-qgcc_cpp_stdinc</code> . Otherwise it is not defined.
<code>__GXX_WEAK__</code>	Defined to 0 on this platform. This macro is set by default.
<code>__HHW_MACOS__</code>	Defined to 1 on the Mac OS X platform.
<code>__HOS_MACOS__</code>	Defined to 1, indicating that the host operating system is Mac OS X.
<code>__MACH__</code>	Defined to 1 on this platform to indicate the MACH object model.
<code>__NATURAL_ALIGNMENT__</code>	Defined to 1 on the Mac OS X platform, for compatibility with GCC.
<code>__OPTIMIZE__</code>	Defined to 2 for optimization level <code>-O</code> or <code>-O2</code> , or to 3 for optimization level <code>-O3</code> or higher.
<code>__OPTIMIZE_SIZE__</code>	Defined to 1 if the options <code>-qcompact</code> and <code>-O</code> are set.
<code>__PPC__</code>	Defined to 1.
<code>__SIZE_TYPE__</code>	Defined to the underlying type of <code>size_t</code> on this platform. On this platform, the macro is defined as long unsigned int .
<code>__THW_MACPPC__</code>	Defined to 1 on the Mac OS X platform.
<code>__TOS_MACOS__</code>	Defined to 1, indicating that the target operating system is Mac OS X.
<code>__unix</code>	Defined to 1 on all Unix-like platforms.
<code>__unix__</code>	Defined to 1 on all Unix-like platforms.
<code>__VEC__</code>	Defined to 10205 by the compiler if AltiVec support is enabled, indicating that the compiler implements the AltiVec Programming Interface model.

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.

Name	Prototype	Description
<code>__bcopy</code>	<code>void __bcopy(char *, char *, int);</code>	Block copy
<code>__bzero</code>	<code>void __bzero(void *, size_t);</code>	Block zero
<code>__check_lock_mp</code>	<code>unsigned int __check_lock_mp (const int* <i>addr</i>, int <i>old_value</i>, int <i>new_value</i>)</code>	<p>Check Lock on MultiProcessor systems.</p> <p>Conditionally updates a single word variable atomically. <i>addr</i> specifies the address of the single word variable. <i>old_value</i> specifies the old value to be checked against the value of the single word variable. <i>new_value</i> specifies the new value to be conditionally assigned to the single word variable. The word variable must be aligned on a full word boundary.</p> <p>Return values:</p> <ol style="list-style-type: none">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
<p><code>__check_lockd_mp</code></p>	<p>unsigned int __check_lock_mp (const long long int* <i>addr</i>, long long int <i>old_value</i>, long long int <i>new_value</i>)</p>	<p>Check Lock Doubleword on MultiProcessor systems.</p> <p>Conditionally updates a double word variable atomically. <i>addr</i> specifies the address of the double word variable. <i>old_value</i> specifies the old value to be checked against the value of the double word variable. <i>new_value</i> 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.</p> <p>Return values:</p> <ol style="list-style-type: none"> 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.
<p><code>__check_lock_up</code></p>	<p>unsigned int __check_lock_up (const int* <i>addr</i>, int <i>old_value</i>, int <i>new_value</i>)</p>	<p>Check Lock on UniProcessor systems.</p> <p>Conditionally updates a single word variable atomically. <i>addr</i> specifies the address of the single word variable. <i>old_value</i> specifies the old value to be checked against the value of the single word variable. <i>new_value</i> specifies the new value to be conditionally assigned to the single word variable. The word variable must be aligned on a full word boundary.</p> <p>Return values:</p> <ul style="list-style-type: none"> • 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_lock_up (const long long int* <i>addr</i> , long long int <i>old_value</i> , int long long <i>new_value</i>)	<p>Check Lock Doubleword on UniProcessor systems.</p> <p>Conditionally updates a double word variable atomically. <i>addr</i> specifies the address of the double word variable. <i>old_value</i> specifies the old value to be checked against the value of the double word variable. <i>new_value</i> 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.</p> <p>Return values:</p> <ul style="list-style-type: none"> • 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* <i>addr</i> , int <i>value</i>)	<p>Clear Lock on MultiProcessor systems.</p> <p>Atomic store of the <i>value</i> into the single word variable at the address <i>addr</i>. The word variable must be aligned on a full word boundary.</p>
__clear_lockd_mp	void __clear_lock_mp (const long long int* <i>addr</i> , long long int <i>value</i>)	<p>Clear Lock Doubleword on MultiProcessor systems.</p> <p>Atomic store of the <i>value</i> into the double word variable at the address <i>addr</i>. The double word variable must be aligned on a double word boundary.</p>
__clear_lock_up	void __clear_lock_up (const int* <i>addr</i> , int <i>value</i>)	<p>Clear Lock on UniProcessor systems.</p> <p>Atomic store of the <i>value</i> into the single word variable at the address <i>addr</i>. The word variable must be aligned on a full word boundary.</p>

Name	Prototype	Description
<code>__clear_lockd_up</code>	<code>void __clear_lock_up (const long long int* <i>addr</i>, long long int <i>value</i>)</code>	Clear Lock Doubleword on UniProcessor systems. Atomic store of the <i>value</i> into the double word variable at the address <i>addr</i> . The double word variable must be aligned on a double word boundary.
<code>__cntlz4</code>	<code>unsigned int __cntlz4(unsigned int);</code>	Count Leading Zeros, 4-Byte Integer
<code>__cntlz8</code>	<code>unsigned int __cntlz8(unsigned long long);</code>	Count Leading Zeros, 8-Byte Integer
<code>__cnttz4</code>	<code>unsigned int __cnttz4(unsigned int);</code>	Count Trailing Zeros, 4-Byte Integer
<code>__cnttz8</code>	<code>unsigned int __cnttz8(unsigned long long);</code>	Count Trailing Zeros, 8-Byte Integer
<code>__dcbt()</code>	<code>void __dcbt (void *);</code>	Data Cache Block Touch. Loads the block of memory containing the specified address into the data cache.
<code>__dcbz()</code>	<code>void __dcbz (void *);</code>	Data Cache Block set to Zero. Sets the specified address in the data cache to zero (0).
<code>__eieio</code>	(Compiler will recognize __eieio built-in.)	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.
<code>__exp</code>	<code>double __exp(double);</code>	Returns the exponential value
<code>__fabs</code>	<code>double __fabs(double);</code>	Returns the absolute value
<code>__fabss</code>	<code>float __fabss(float);</code>	Returns the short floating point absolute value
<code>__fcfid</code>	<code>double __fcfid (double)</code>	Floating Convert From Integer Doubleword. The 64bit signed fixedpoint operand is converted to a double-precision floating-point.

Name	Prototype	Description
__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).
__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 __fmsub(double, double, double);	Floating point multiply-add
__fmadds	float __fmsub(float, float, float);	Floating point multiply-add
__fmsub	double __fmsub(double, double, double);	Floating point multiply-sub
__fmsubs	float __fmsub(float, float, float);	Floating point multiply-sub
__fnabs	double __fnabs(double);	Floating point negative absolute
__fnabss	float __fnabss(float);	Floating point short negative absolute
__fnmadd	double __fnmadd(double, double, double);	Floating point negative multiply-add
__fnmadds()	float __fnmadds (float, float, float);	Floating point negative multiply-add

Name	Prototype	Description
<code>__fnmsub</code>	<code>double __fnmsub(double, double, double);</code>	Floating point negative multiply-sub
<code>__fnmsubs()</code>	<code>float __fnmsubs (float, float, float);</code>	<code>__fnmsubs (a, x, y) = [- (a * x - y)]</code>
<code>__fres()</code>	<code>float __fres (float);</code>	<code>__fres (x) = [(estimate of) 1.0/x]</code>
<code>__frsqtrte()</code>	<code>double __frsqtrte (double);</code>	<code>__frsqtrte (x) = [(estimate of) 1.0/sqrt(x)]</code>
<code>__fsel()</code>	<code>double __fsel (double, double, double);</code>	if ($a \geq 0.0$) then <code>__fsel (a, x, y) = x</code> ; else <code>__fsel (a, x, y) = y</code>
<code>__fsels()</code>	<code>float __fsels (float, float, float);</code>	if ($a \geq 0.0$) then <code>__fsels (a, x, y) = x</code> ; else <code>__fsels (a, x, y) = y</code>
<code>__fsqrt()</code>	<code>double __fsqrt (double);</code>	<code>__fsqrt (x) = square root of x</code>
<code>__fsqrts()</code>	<code>float __fsqrts (float);</code>	<code>__fsqrts (x) = square root of x</code>
<code>__iospace_eieio</code>	<code>void __iospace_eieio(void);</code>	Generates an EIEIO instruction
<code>__iospace_lwsync</code>	(equivalent to: <code>void __iospace_lwsync(void);</code>)	Generates a lwsync instruction
<code>__iospace_sync</code>	(equivalent to: <code>void __iospace_sync(void);</code>)	Generates a sync instruction
<code>__isync</code>	<code>void __isync(void);</code>	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.
<code>__load2r</code>	<code>unsigned short __load2r(unsigned short*);</code>	Load halfword byte reversed
<code>__load4r</code>	<code>unsigned int __load4r(unsigned int*);</code>	Load word byte reversed
<code>__lwsync</code>	(Compiler will recognize <code>__lwsync</code> built-in.)	Extra name for the existing <code>__iospace_lwsync</code> built-in. Compiler will recognize <code>__lwsync</code> built-in. Everything except for the name is exactly same as for <code>__iospace_lwsync</code> . <code>__lwsync</code> is consistent with the corresponding PowerPC instruction name. This function is supported only by the PowerPC 970 processor.
<code>__mtfsb0</code>	<code>void __mtfsb0(unsigned int bt)</code>	Move to FPSCR Bit 0. Bit <i>bt</i> of the FPSCR is set to 0. <i>bt</i> must be a constant and $0 \leq bt \leq 31$.

Name	Prototype	Description
<code>__mtfsb1</code>	<code>void __mtfsb1(unsigned int <i>bt</i>)</code>	Move to FPSCR Bit 1. Bit <i>bt</i> of the FPSCR is set to 1. <i>bt</i> must be a constant and $0 \leq bt \leq 31$.
<code>__mtfsf</code>	<code>void __mtfsf(unsigned int <i>flm</i>, unsigned int <i>frb</i>)</code>	Move to FPSCR Fields. The contents of <i>frb</i> are placed into the FPSCR under control of the field mask specified by <i>flm</i> . The field mask <i>flm</i> identifies the 4bit fields of the FPSCR affected. <i>flm</i> must be a constant 8-bit mask.
<code>__mtfsfi</code>	<code>void __mtfsfi(unsigned int <i>bf</i>, unsigned int <i>u</i>)</code>	Move to FPSCR Field Immediate. The value of the <i>u</i> is placed into FPSCR field specified by <i>bf</i> . <i>bf</i> and <i>u</i> must be constants, with $0 \leq bf \leq 7$ and $0 \leq u \leq 15$.
<code>__mulhd</code>	<code>long long int __mulhd(long long int <i>ra</i>, long long int <i>rb</i>)</code>	Multiply High Doubleword Signed. Returns the highorder 64 bits of the 128bit product of the operands <i>ra</i> and <i>rb</i> .
<code>__mulhdu</code>	<code>unsigned long long int __mulhdu(unsigned long long int <i>ra</i>, unsigned long long int <i>rb</i>)</code>	Multiply High Doubleword Unsigned. Returns the highorder 64 bits of the 128bit product of the operands <i>ra</i> and <i>rb</i> .
<code>__mulhw</code>	<code>int __mulhw(int <i>ra</i>, int <i>rb</i>)</code>	Multiply High Word Signed. Returns the highorder 32 bits of the 64bit product of the operands <i>ra</i> and <i>rb</i> .
<code>__mulhwu</code>	<code>unsigned int __mulhwu(unsigned int <i>ra</i>, unsigned int <i>rb</i>)</code>	Multiply High Word Unsigned. Returns the highorder 32 bits of the 64bit product of the operands <i>ra</i> and <i>rb</i> .
<code>__parthds</code>	<code>int __parthds(void);</code>	Returns the value of the <code>parthds</code> run-time option. If the <code>parthds</code> option is not explicitly set by the user, the function returns the default value set by the run-time library. If the <code>-qsmp</code> compiler option was not specified during program compilation, this function returns 1 regardless of run-time options selected.
<code>__pow</code>	<code>double __pow(double, double);</code>	

Name	Prototype	Description
__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
__rdlam	unsigned long long __rdlam(unsigned long long <i>rs</i> , unsigned int <i>shift</i> , unsigned long long <i>mask</i>)	Rotate Double Left and AND with Mask. The contents of <i>rs</i> are rotated left <i>shift</i> bits. The rotated data is ANDed with the mask and returned as a result. <i>mask</i> must be a constant and represent a contiguous bitfield.
__readflm	double __readflm();	Read floating point status/control register
__rldimi	unsigned long long __rldimi(unsigned long long <i>rs</i> , unsigned long long <i>is</i> , unsigned int <i>shift</i> , unsigned long long <i>mask</i>)	Rotate Left Doubleword Immediate then Mask Insert. Rotates <i>rs</i> left <i>shift</i> bits then inserts <i>rs</i> into <i>is</i> under bit mask <i>mask</i> . Shift must be a constant and $0 \leq \text{shift} \leq 63$. <i>mask</i> must be a constant and represent a contiguous bitfield.
__rlwimi	unsigned int __rlwimi(unsigned int <i>rs</i> , unsigned int <i>is</i> , unsigned int <i>shift</i> , unsigned int <i>mask</i>)	Rotate Left Word Immediate then Mask Insert. Rotates <i>rs</i> left <i>shift</i> bits then inserts <i>rs</i> into <i>is</i> under bit mask <i>mask</i> . Shift must be a constant and $0 \leq \text{shift} \leq 31$. <i>mask</i> must be a constant and represent a contiguous bitfield.
__rlwnm	unsigned int __rlwnm(unsigned int <i>rs</i> , unsigned int <i>shift</i> , unsigned int <i>mask</i>)	Rotate Left Word then AND with Mask. Rotates <i>rs</i> left <i>shift</i> bits, then ANDs <i>rs</i> with bit mask <i>mask</i> . <i>mask</i> must be a constant and represent a contiguous bitfield.
__rotatel4	unsigned int __rotatel4(unsigned int <i>rs</i> , unsigned int <i>shift</i>)	Rotate Left Word. Rotates <i>rs</i> left <i>shift</i> bits.
__setflm	double __setflm(double);	Set Floating Point Status/Control Register
__setrnd	double __setrnd(int);	Set Rounding Mode
__stfiw	void __stfiw(const int* <i>addr</i> , double <i>value</i>);	Store Floating-Point as Integer Word. The contents of the loworder 32 bits of <i>value</i> are stored, without conversion, into the word in storage addressed by <i>addr</i> .
__store2r	void __store2r(unsigned short, unsigned short*);	Store 2-byte register

Name	Prototype	Description
__store4r	void __store2r(unsigned int, unsigned int *);	Store 4-byte register
__sync	(Compiler will recognize __sync built-in.)	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.
__tdw	void __tdw(long long <i>a</i> , long long <i>b</i> , unsigned int <i>TO</i>)	Trap Doubleword. Operand <i>a</i> is compared with operand <i>b</i> . This comparison results in five conditions which are ANDed with a 5-bit constant <i>TO</i> containing a value of 0 to 31 inclusive. If the result is not 0 the system trap handler is invoked. Each bit position, if set, indicates one or more of the following possible conditions: 0 (high-order bit) <i>a</i> Less Than <i>b</i> , using signed comparison. 1 <i>a</i> Greater Than <i>b</i> , using signed comparison. 2 <i>a</i> Equal <i>b</i> 3 <i>a</i> Less Than <i>b</i> , using unsigned comparison. 4 (low order bit) <i>a</i> Greater Than <i>b</i> , using unsigned comparison.
__trap	void __trap(int);	Trap
__trapd	void __trapd (longlong);	Trap if the parameter is not zero.

Name	Prototype	Description
__tw	void __tw(int <i>a</i> , int <i>b</i> , unsigned int <i>TO</i>)	<p>Trap Word.</p> <p>Operand <i>a</i> is compared with operand <i>b</i>. This comparison results in five conditions which are ANDed with a 5-bit constant <i>TO</i> containing a value of 0 to 31 inclusive.</p> <p>If the result is not 0 the system trap handler is invoked. Each bit position, if set, indicates one or more of the following possible conditions:</p> <p>0 (high-order bit) <i>a</i> Less Than <i>b</i>, using signed comparison.</p> <p>1 <i>a</i> Greater Than <i>b</i>, using signed comparison.</p> <p>2 <i>a</i> Equal <i>b</i></p> <p>3 <i>a</i> Less Than <i>b</i>, using unsigned comparison.</p> <p>4 (low order bit) <i>a</i> Greater Than <i>b</i>, using unsigned comparison.</p>
__usrthds	int __usrthds(void);	<p>Returns the value of the usrthds run-time option.</p> <p>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.</p>

Appendix C. Libraries in XL C/C++ Advanced Edition for Mac OS X

Redistributable Libraries

XL C/C++ Advanced Edition for Mac OS X 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++ Advanced Edition for Mac OS X.

libibmc++.dylib

Used only by C++ programs.

Order of Linking

XL C/C++ Advanced Edition for Mac OS X links libraries in the following order:

1. user .o files and libraries
2. XL C/C++ Advanced Edition for Mac OS X 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 17 for more information about compiler default configuration files in general.

ld Command Components	Options		ld Arguments	xldriver attributes
	xlC/xlc++/xlC	gcc/g++		
ld	Default	Default	ld	ld
	-qmksprobj	-dynamiclib	libtool	libtool
arch	Default	Default	-arch ppc	Option added to command line by xldriver
	-qmksprobj	-dynamiclib	-arch_only ppc	Option added to command line by xldriver
output kind	Default	Default	-dynamic	Option -dynamic added to command line by xldriver
	-qmksprobj	-dynamiclib	-dynamic -noall_load	Option -dynamic and -noall_load added to command line by xldriver
call to main()	Default	Default	-lcrt -lcrtbegin	crt
	-qmksprobj	-dynamiclib		
	-p	-p	-lcrt -lcrtbegin	mcrt
-pG	-pG	-lgcrt1.o -lcrt2.o	/usr/lib/crti.o	gcrt

Id Command Components	Options		ld Arguments	xldriver attributes
	xlC/xlC++/xlC	gcc/g++		
-qnocrt	-nostartfiles	No standard system startup files		
library search paths			-L/usr/lib/gcc/darwin/3.3, -L/usr/lib/gcc/darwin, -L/usr/libexec/gcc/darwin/ppc/3.3/../../..	gcc_libdirs
VAC/C++ library search paths				libraries libdirs
g++ required standard libraries	Default	Default	-lstdc++, -lgcc, -lSystem	gcc_libs
	-qnolib	-nodefaultlibs	No standard system libraries	
		-nostdlib		
gcc required standard libraries	Default	Default	-lgcc -lSystem	gcc_libs
	-qnolib	-nodefaultlibs	No standard system libraries	
		-nostdlib		

Appendix D. Problem Solving

Topics in this section are:

- “Message Catalog Errors”
- “Correcting Paging Space Errors During Compilation” on page 314

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 all nondiagnostic 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++ Advanced Edition for Mac OS X internal message number. This message is issued in English only.

To determine message catalogs which are installed on your system, list all of the file names for the catalogs 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/6.0/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 11

“Set Other Environment Variables” on page 11

Correcting Paging Space Errors During Compilation

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

```
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++ Advanced Edition for Mac OS X uses the American National Standard Code for Information Interchange (ASCII) character set as its collating sequence.

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	0C	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- <u></u>	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).



SC09-7866-00

