

z/OS
Version 2 Release 4

*Compiler Reference
for XL C/C++ V2.4.1 for z/OS V2.4*



Note

Before using this information and the product it supports, read the information in [“Notices” on page 121.](#)

This edition applies to Version 2 Release 4 Modification 1 of XL C/C++ for IBM® z/OS® (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Last updated: 2019-12-19

© **Copyright International Business Machines Corporation 2019.**

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

Contents

About this document.....	vii
Where to find more information.....	x
z/OS Basic Skills in IBM Knowledge Center.....	x
Technical support.....	xi
How to send your comments to IBM.....	xi
If you have a technical problem.....	xi
 Chapter 1. Compiling and linking applications.....	1
Invoking the compiler.....	1
Command-line syntax.....	1
Types of input files.....	2
Types of output files.....	3
Specifying compiler options.....	3
Specifying compiler options on the command line.....	4
Specifying compiler options in a configuration file.....	4
Specifying compiler options in program source files.....	4
Resolving conflicting compiler options.....	5
Preprocessing.....	5
Directory search sequence for included files.....	6
Linking.....	7
Order of linking.....	8
Compiler messages and listings.....	8
Compiler messages.....	8
Return codes.....	9
Compiler listings.....	10
 Chapter 2. Configuring compiler defaults.....	11
Setting environment variables.....	11
Compile-time and link-time environment variables.....	11
Runtime environment variables.....	11
Using custom compiler configuration files.....	11
Creating custom configuration files.....	12
Editing the default configuration file.....	14
Configuration file attributes.....	14
 Chapter 3. Compiler options reference.....	17
Summary of compiler options by functional category.....	17
Output control.....	17
Input control.....	18
Language element control.....	18
Template control (C++ only).....	19
Floating-point and integer control.....	20
Object code control.....	20
Error checking and debugging.....	20
Listings, messages, and compiler information.....	23
Optimization and tuning.....	24
Linking.....	25
Portability and migration.....	25
Compiler customization.....	25
Individual option descriptions.....	26

-# (pound sign).....	27
++ (plus sign) (C++ only).....	28
-C.....	28
-c.....	29
-D.....	30
-E.....	31
-e.....	32
-F.....	33
-g.....	34
-I.....	34
-L.....	35
-l.....	36
-M.....	37
-MD.....	38
-MF.....	39
-MG.....	40
-MM.....	41
-MMD.....	41
-MQ.....	42
-MT.....	43
-O, -qoptimize.....	44
-o.....	46
-P.....	47
-r.....	48
-s.....	49
-U.....	49
-v, -V.....	50
-W.....	51
-qansialias.....	52
-qarch.....	54
-qascii.....	55
-qasm (-fasm).....	56
-qasmlib.....	57
-qassert.....	58
-qchars (-fsigned-char, -funsigned-char).....	59
-qcompact.....	60
-qcompress.....	60
-qcsect.....	61
-qdebug.....	63
-qdigraph.....	67
-qeh (C++ only).....	67
-qexportall.....	68
-qfloat.....	69
-qgonumber.....	70
-qhalt.....	71
-qignerrno.....	72
-qinclude.....	73
-qinline.....	73
-qlanglvl (-std).....	75
-qlibansi.....	77
-qlist.....	78
-qmakedep.....	79
-qmaxmem.....	80
-qmemory.....	81
-qoffset.....	82
-qoptfile.....	83
-qphaseid, -qphsinfo.....	83
-qro.....	85

-qroconst.....	85
-qrtccheck.....	86
-qrtti (-ftrtti) (C++ only).....	87
-qservice.....	88
-qshowmacros.....	89
-qspill.....	90
-qstackprotect.....	91
-qstrict.....	92
-qstrict_induction.....	93
-qsyntaxonly (-fsyntax-only).....	94
-qtemplatedepth (-ftemplate-depth) (C++ only).....	95
-qthreaded.....	96
-qtune.....	96
Supported GCC options.....	98
Chapter 4. Compiler pragmas reference.....	101
Pragma directive syntax.....	101
Scope of pragma directives.....	101
Supported IBM pragmas.....	102
#pragma convert.....	102
#pragma csect.....	103
#pragma execution_frequency.....	104
#pragma linkage (C only).....	105
#pragma leaves.....	107
#pragma map.....	107
#pragma option_override.....	109
#pragma priority (C++ only).....	110
#pragma weak (C only).....	111
#pragma reachable.....	112
Chapter 5. Compiler predefined macros.....	115
General macros.....	115
Macros indicating the z/OS XL C/C++ compiler.....	116
Macros related to the platform.....	116
Macros related to compiler features.....	117
Macros related to compiler option settings.....	117
Macros related to language levels.....	118
Notices.....	121
Trademarks.....	121
Standards.....	122
Index.....	123

About this document

This information supports IBM z/OS XL C/C++ (5650-ZOS) and contains information about IBM XL C/C++ V2.4.1 for z/OS V2.4.

Note: This publication refers to IBM XL C/C++ V2.4.1 for z/OS V2.4 as XL C/C++ V2.4.1.

This document is a reference for the XL C/C++ V2.4.1 compiler. Although it provides information about compiling and linking applications written in C and C++, it is primarily intended as a reference for compiler command-line options, pragma directives, predefined macros, and environment variables.

Who should read this document

This document is for experienced C or C++ developers who have some familiarity with the z/OS XL C/C++ compiler or other command-line compilers on z/OS operating systems. It assumes thorough knowledge of the C or C++ programming language and basic knowledge of operating system commands. Although this information is intended as a reference guide, programmers new to z/OS XL C/C++ can still find information about the capabilities and features unique to the XL C/C++ V2.4.1 compiler.

How to use this document

You can use this document to:

- Help determine whether and how you can continue to use existing source code, object code, and load modules
- Become aware of the changes in compiler and runtime behavior that may affect your migration from earlier versions of the compiler

Note: In most situations, existing well-written applications can continue to work without modification.

This document does not:

- Discuss all of the enhancements that have been made to the z/OS XL C/C++ compiler.

While this document covers topics such as configuring the compiler environment, and compiling and linking C or C++ applications using the XL C/C++ V2.4.1 compiler, it does not include the following topics:

- Compiler installation and system requirements: see the [Program Directory for XL C/C++ V2R4M1 web deliverable for z/OS](http://publibfp.dhe.ibm.com/epubs/pdf/i1357010.pdf) (<http://publibfp.dhe.ibm.com/epubs/pdf/i1357010.pdf>).
- Basic install and run information: see [Getting Started with XL C/C++ V2.4.1 for z/OS V2.4](#).

Typographical conventions

The following table explains the typographical conventions used in this document.

Table 1. Typographical conventions		
Typeface	Indicates	Example
bold	Commands, executable names, compiler options and pragma directives that contain lower-case letters.	The xlcclang invocation command invokes the XL C/C++ V2.4.1 compiler.
<i>italics</i>	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	Make sure that you update the <i>size</i> parameter if you return more than the <i>size</i> requested.

Table 1. Typographical conventions (continued)		
Typeface	Indicates	Example
<u>underlining</u>	The default setting of a parameter of a compiler option or directive.	<u>nomaf</u> maf
monospace	Programming keywords and library functions, compiler built-in functions, file and directory names, examples of program code, command strings, or user-defined names.	If one or two cases of a switch statement are typically executed much more frequently than other cases, break out those cases by handling them separately before the switch statement.

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.

For users accessing IBM Knowledge Center using a screen reader, syntax diagrams are provided in dotted decimal format.

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 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.

Note: If a syntax diagram shows a character that is not alphanumeric (for example, parentheses, periods, commas, equal signs, a blank space), enter the character as part of the syntax.

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.

The following table provides syntax examples.

Table 2. Syntax examples

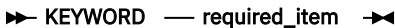
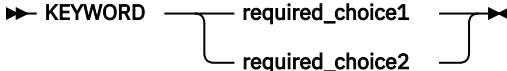
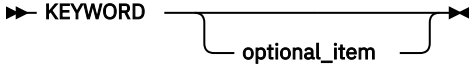
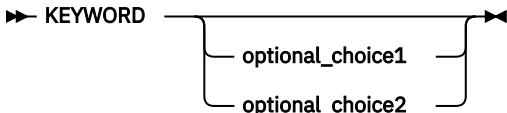
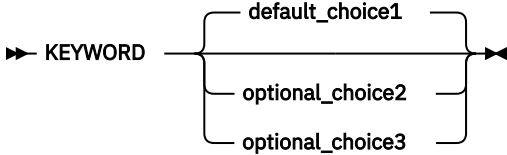
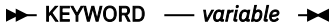
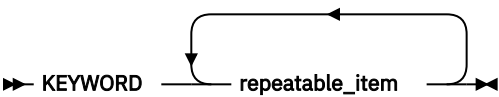
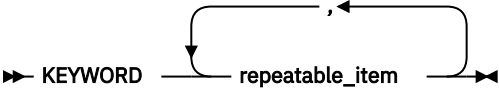
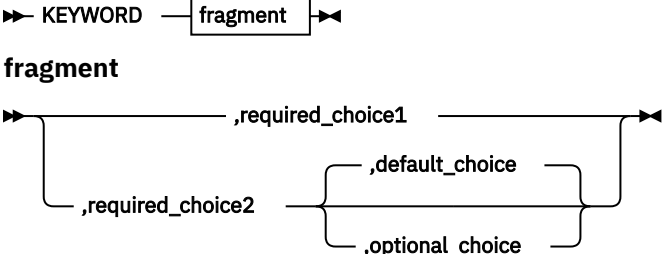
Item	Syntax example
Required item. Required items appear on the main path of the horizontal line. You must specify these items.	
Required choice. A required choice (two or more items) appears in a vertical stack on the main path of the horizontal line. You must choose one of the items in the stack.	
Optional item. Optional items appear below the main path of the horizontal line.	
Optional choice. An optional choice (two or more items) appears 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 2. Syntax examples (continued)

Item	Syntax example
Repeatable item. An arrow returning to the left above the main path of the horizontal line indicates an item that can be repeated. A character within the arrow means you must separate repeated items with that character.	 
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.	
Fragment. The fragment symbol indicates that a labeled 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.	

Softcopy documents

The XL C/C++ V2.4.1 publications are supplied in PDF format and available for download from the z/OS XL C/C++ documentation library (<https://www.ibm.com/support/pages/zos-xl-cc-documentation-library>).

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

Where to find more information

For an overview of the information associated with z/OS, see z/OS Information Roadmap (https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.4.0/com.ibm.zos.v2r4.e0zc100/abstract.htm).

Additional information on z/OS XL C/C++ is available on the [Marketplace page for z/OS XL C/C++ \(www.ibm.com/us-en/marketplace/xl-cpp-compiler-zos\)](http://www.ibm.com/us-en/marketplace/xl-cpp-compiler-zos).

z/OS Basic Skills in IBM Knowledge Center

z/OS Basic Skills in IBM Knowledge Center is a Web-based information resource intended to help users learn the basic concepts of z/OS, the operating system that runs most of the IBM mainframe computers in use today. IBM Knowledge Center is designed to introduce a new generation of Information Technology professionals to basic concepts and help them prepare for a career as a z/OS professional, such as a z/OS system programmer.

Specifically, z/OS Basic Skills is intended to achieve the following objectives:

- Provide basic education and information about z/OS without charge
- Shorten the time it takes for people to become productive on the mainframe
- Make it easier for new people to learn z/OS.

is available to all users (no login required).

Technical support

Additional technical support is available from the [z/OS XL C/C++ Support page \(www.ibm.com/mysupport/s/topic/OTO0z0000006v6TGAQ/xl-cc\)](http://www.ibm.com/mysupport/s/topic/OTO0z0000006v6TGAQ/xl-cc). This page provides a portal with search capabilities to technical support FAQs and other support documents.

For the latest information about XL C/C++ V2.4.1 for z/OS V2.4, visit [Marketplace page for z/OS XL C/C++ \(www.ibm.com/us-en/marketplace/xl-cpp-compiler-zos\)](http://www.ibm.com/us-en/marketplace/xl-cpp-compiler-zos).

If you cannot find what you need, you can e-mail:

compinfo@cn.ibm.com

How to send your comments to IBM

We appreciate your input on this documentation. Please provide us with any feedback that you have, including comments on the clarity, accuracy, or completeness of the information.

You can send an email to compinfo@cn.ibm.com and include the following information:

- Your name and address
- Your email address
- Your phone or fax number
- The publication title and order number:
 Compiler Reference for XL C/C++ V2.4.1 for z/OS V2.4
 SC31-5801-00
- The topic and page number or URL of the specific information to which your comment relates
- The text of your comment.

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

IBM or any other organizations use the personal information that you supply to contact you only about the issues that you submit.

If you have a technical problem

If you have a technical problem, take one or more of the following actions:

- Visit the [IBM Support Portal \(support.ibm.com\)](http://support.ibm.com).
- Contact your IBM service representative.
- Call IBM technical support.

Chapter 1. Compiling and linking applications

By default, when you invoke the XL C/C++ V2.4.1 compiler, all of the following phases of translation are performed:

- Preprocessing of program source
- Compiling and assembling into object files

These different translation phases are actually performed by separate executables, which are referred to as compiler *components*. However, you can use compiler options to perform only certain phases, such as preprocessing, or assembling. You can then re-invoke the compiler to resume compiling preprocessed output.

The following sections describe how to invoke the XL C/C++ V2.4.1 compiler to preprocess or compile C and C++ sources and link object files:

- [“Invoking the compiler” on page 1](#)
- [“Types of input files” on page 2](#)
- [“Types of output files” on page 3](#)
- [“Specifying compiler options” on page 3](#)
- [“Preprocessing” on page 5](#)
- [“Linking” on page 7](#)

Invoking the compiler

The XL C/C++ V2.4.1 compiler can be invoked by using the **xlclang** invocation command to compile and link C programs and the **xlclang++** invocation command to compile and link C++ programs.

The compiler configuration file defines default option settings and, in some cases, macros; for information about the defaults implied by a particular invocation, see the `/usr/lpp/cbclib/xlclang/etc/xlclang.cfg` file.

Notes:

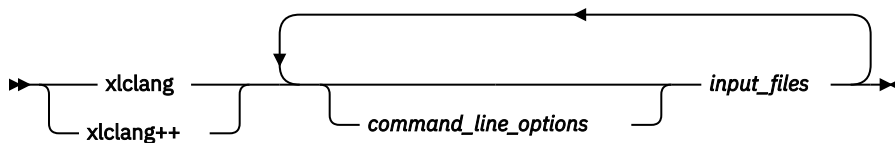
- When you install the XL C/C++ V2.4.1 compiler, a default configuration file is automatically generated during the installation procedure under `/usr/lpp/cbclib/xlclang/sample/xlclang.cfg`, which can be customized to match your installation environment. Then you install the customized configuration file `xlclang.cfg` in the `/usr/lpp/cbclib/xlclang/etc/` directory, which is searched by the **xlclang** utility for the default configuration file.
- A future PTF will only update the default configuration file `xlclang.cfg` under `/usr/lpp/cbclib/xlclang/sample/`. You need to apply the updates to your customized `xlclang.cfg` file under the `/usr/lpp/cbclib/xlclang/etc/` directory.

Related information

- [“-qlanglvl \(-std\)” on page 75](#)

Command-line syntax

You invoke the compiler using the following syntax:



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

In addition to invoking the C/C++ compiler, the **xlclang** and **xlclang++** commands can also invoke Assembler and Binder, similar to the **xlc** and **xlc** commands. For detailed information, see "z/OS XL C/C++ User's Guide".

Your program can consist of several input files. All of these source files can be compiled at once using only one command line of the compiler. Although more than one source file can be compiled using a single command line of the compiler, you can specify only one set of compiler options on the command line per invocation. Each distinct set of command-line compiler options that you want to specify requires a separate invocation.

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

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

Related information

- [“Types of input files” on page 2](#)

Types of input files

The compiler processes the source files in the order in which they are specified on the command line. 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 binder does not run.

By default, the compiler preprocesses and compiles all the specified source files. Although you usually want to use this default, you can use the compiler to preprocess the source file without compiling; see [“Preprocessing” on page 5](#) for details.

You can input the following types of files to the XL C/C++ V2.4.1 compiler:

C and C++ source files

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

To use the compiler to compile a C language source file, the source file must have a **.c** (lowercase c) suffix.

To use the compiler to compile a C++ language source file, the source file must have a **.C** (uppercase C), **.cc**, **.cpp**, or **.cxx** suffix.

Preprocessed source files

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

Object files

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

Assembler files

Assembler files must have a `.s` suffix, for example, *file_name.s*. Assembler files are assembled to create an object file.

Definition side decks

Definition side decks provide input to the binder when linking an application that refers to functions or variables defined in a DLL. A definition side deck is a file with suffix `.x`, which contains input that allows the binder to resolve references to symbols exported from a DLL.

Related information

[“Input control” on page 18](#)

Types of output files

You can specify the following types of output files when invoking the XL C/C++ V2.4.1 compiler:

Executable files

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

Object files

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

You can link the object files later into a single executable file by a separate command invocation.

Definition side decks

If you specify the `-qexportall` compiler option or the `_Export` keyword in the source files, the compiler produces a definition side deck, which has a suffix `.x` and is used to resolve references to exported symbols in an application that uses this DLL.

Listing files

If you specify `-qlist=`, a compiler listing file, *file_name.lst*, is produced for each input file. The listing file is placed in your current directory.

Make dependency files

If you specify the `-qmakedep`, `-M`, `-MD`, `-MM`, or `-MMD` option, a make dependency file suitable for inclusion in a makefile, *file_name.d* is produced for each input file.

Related information

[“Output control” on page 17](#)

Specifying compiler options

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

- [On the command line](#)
- [In a custom configuration file](#), which is a file with a `.cfg` extension
- [In your source program](#)

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

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

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

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

Option conflicts that do not follow this priority sequence are described in [“Resolving conflicting compiler options”](#) on page 5.

Related information

[“Compiler options reference”](#) on page 17

[“Compiler pragmas reference”](#) on page 101

Specifying compiler options on the command line

Most options specified on the command line override both the default settings of the option and options set in the configuration file. Similarly, most options specified on the command line are in turn overridden by pragma directives, which provide you a means of overriding compiler options right in the source file. Options that do not follow this scheme are listed in [“Resolving conflicting compiler options”](#) on page 5.

Specifying compiler options in a configuration file

The default configuration file (/usr/lpp/cbclib/xlclang/etc/xlclang.cfg) defines values and compiler options for the compiler. The compiler refers to this file when compiling C or C++ programs.

The configuration file is a plain text file. You can edit this file, or create an additional customized configuration file to support specific compilation requirements. For more information, see [“Using custom compiler configuration files”](#) on page 11.

Specifying compiler options in program source files

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

- Using **#pragma name** syntax

Some options also have corresponding pragma directives that use a pragma-specific syntax, which may include additional or slightly different suboptions. Throughout the section “Individual option descriptions” on page 26, each option description indicates whether this form of the pragma is supported, and the syntax is provided.

- Using the standard C99 `_Pragma` operator

For options that support either forms of the pragma directives listed above, you can also use the C99 `_Pragma` operator syntax in both C and C++.

Complete details on pragma syntax are provided in [“Pragma directive syntax”](#) on page 101.

Other pragmas do not have equivalent command-line options; these are described in detail throughout Chapter 4, [“Compiler pragmas reference,”](#) on page 101.

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

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

Resolving conflicting compiler options

In general, if more than one variation of the same option is specified, 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. However, some options have cumulative effects when they are specified more than once; examples are the **-I***directory* and **-L***directory* options.

When options such as **-qdebug**, **-qfloat**, and **-qstrict** are specified with suboptions for multiple times, each suboption overrides previous specifications of that suboption, but different suboptions are cumulative.

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

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

Not all option conflicts are resolved using the preceding rules. The following table summarizes exceptions and how the compiler handles conflicts between them.

Option	Conflicting options	Resolution
-E	-P	-E
-P	-C , -O	-P
-#	-v	-#

Preprocessing

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

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

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

Option	Description
“-E” on page 31	Preprocesses the source files and writes the output to standard output. By default, <code>#line</code> directives are generated.
“-P” on page 47	Preprocesses the source files and creates an intermediary file with a <code>.i</code> file name suffix for each source file. By default, <code>#line</code> directives are not generated.
“-C” on page 28	Preserves comments in preprocessed output.
“-D” on page 30	Defines a macro name from the command line, as if in a <code>#define</code> directive.

Option	Description
“-qmakedep” on page 79	Produces the dependency files that are used by the make tool for each source file.
“-M” on page 37 ¹	Generates a rule suitable for the make tool that describes the dependencies of the input file.
-MD ¹	Compiles the source files, generates the object file, and generates a rule suitable for the make tool that describes the dependencies of the input file in a file with the name of the input file and suffix .d.
-MF <i>file</i> ¹	Specifies the file to write the dependencies to. The -MF option must be specified with option -M or -MM .
-MG ¹	Assumes that missing header files are generated files and adds them to the dependency list without raising an error. The -MG option must be used with option -M , -MD , -MM , or -MMD .
-MM ¹	Generates a rule suitable for the make tool that describes the dependencies of the input file, but does not mention header files that are found in system header directories nor header files that are included from such a header.
-MMD ¹	Compiles the source files, generates the object file, and generates a rule suitable for the make tool that describes the dependencies of the input file in a file with the name of the input file and suffix .d. However, the dependencies do not include header files that are found in system header directories nor header files that are included from such a header.
-MP ¹	Instructs the C preprocessor to add a phony target for each dependency other than the input file.
-MQ <i>target</i> ¹	Changes the target of the rule emitted by dependency generation and quotes any characters that are special to the make tool.
-MT <i>target</i> ¹	Changes the target of the rule emitted by dependency generation.
“-U” on page 49	Undefines a macro name defined by the compiler or by the -D option.
Note: 1. For details about the option, see GNU Compiler Collection online documentation (http://gcc.gnu.org/onlinedocs/).	

Directory search sequence for included files

The XL C/C++ V2.4.1 compiler supports the following types of included files:

- Header files supplied by the compiler (referred to throughout this document as *z/OS XL C/C++ headers*)
- Header files mandated by the C and C++ standards (referred to throughout this document as *system headers*)
- Header files supplied by the operating system (also referred to throughout this document as *system headers*)
- User-defined header files

You can use any of the following methods to include any type of header file:

- Use the standard `#include <file_name>` preprocessor directive to include header files that are not user-defined.
- Use the standard `#include "file_name"` preprocessor directive to include user-defined header files.
- Use the [“-qinclude” on page 73](#) compiler option.

If you specify the header file using a full (absolute) path name, you can use these methods interchangeably, regardless of the type of header file you want to include. However, if you specify the header file using a *relative* path name, the compiler uses a different directory search order for locating the file depending on the method used to include the file.

The following summarizes the search order used by the compiler to locate header files depending on the mechanism used to include the files and on the compiler options that are in effect:

1. Header files included with “`-qinclude`” on page 73 only: The compiler searches the current (working) directory from which the compiler is invoked.
2. Header files included with “`-qinclude`” on page 73 or `#include "file_name"`: The compiler searches the directory in which the source file is located.
3. All header files: The compiler searches each directory specified by the `-I` compiler option, in the order that it displays on the command line.
4. All header files: The compiler searches the standard directory for the system headers. The default directory for these headers is specified in the compiler configuration file. This location is set during installation.

Related information

- “`-I`” on page 34
- “`-qinclude`” on page 73
- `-isystem`. For details, see [GNU Compiler Collection online documentation](http://gcc.gnu.org/onlinedocs/)(<http://gcc.gnu.org/onlinedocs/>).

Linking

The binder links specified object files to create one executable file. Invoking the compiler with one of the invocation commands automatically calls the binder unless you specify one of the following compiler options:

- `-c`
- `-E`
- `-M`
- `-P`
- `-S`
- `-fsyntax-only` (`-qsyntaxonly`)
- `-#`

Input files

Object files, unstripped executable files, and library files serve as input to the binder. Object files must have a `.o` suffix, for example, `filename.o`. Static library file names have a `.a` suffix, for example, `filename.a`. DLL definition side file names have a `.x` suffix, for example, `filename.x`.

Output files

The binder generates an *executable file* and places it in your current directory. The default name for an executable file is `a.out`. To name the executable file explicitly, use the `-o file_name` option with the invocation command, where `file_name` is the name you want to give to the executable file. For example, to compile `myfile.c` and generate an executable file called `myfile`, enter:

```
xlclang myfile.c -o myfile
```

The compiler invocation commands set several binder options, and link some standard files into the executable output by default. In most cases, it is better to use one of the compiler invocation commands to link your object files. For a complete list of options available for linking, see “Linking” on page 25.

Note: If you want to use a nondefault binder, you can customize the configuration file of the compiler to use the nondefault binder. For more information about how to customize the configuration file, see [Using custom compiler configuration files](#) and [Creating custom configuration files](#).

Order of linking

The compiler links libraries in the following order:

1. System startup libraries
2. User .o files and libraries
3. XL C/C++ V2.4.1 libraries
4. C++ standard libraries
5. C standard libraries

Related information

- [“Linking” on page 25](#)

Compiler messages and listings

The following sections discuss the various information generated by the compiler after compilation.

- [“Compiler messages” on page 8](#)
- [“Compiler listings” on page 10](#)

Compiler messages

When the compiler encounters a programming error while compiling a C or C++ source program, it issues a diagnostic message to the standard error device. You can control which code constructs cause the compiler to emit errors and warning messages and how they are displayed to the console.

Message severity levels and compiler response

The XL C/C++ V2.4.1 compiler uses a multilevel classification scheme for diagnostic messages. Each level of severity is associated with a compiler response. The table below provides a key to the abbreviations for the severity levels and the associated default compiler response.

You can use the **-qhalt** option to stop the compilation for warnings and all types of errors.

Table 3. Compiler message severity levels			
Letter	Severity	Synonym	Compiler response
I	Informational	note	Compilation continues and object code is generated. The message reports conditions found during compilation.
W	Warning	warning	Compilation continues and object code is generated. The message reports valid but possibly unintended conditions.
E	Error	error	Compilation continues and object code is generated. The compiler can correct the error conditions that are found, but the program might not produce the expected results.

Table 3. Compiler message severity levels (continued)

Letter	Severity	Synonym	Compiler response
S	Severe error	error	<p>Compilation continues, but object code is not generated. The compiler cannot correct the error conditions that are found.</p> <ul style="list-style-type: none"> • If the message indicates a resource limit (for example, file system full), provide additional resources and recompile. • If the message indicates that different compiler options are needed, recompile using those options. • Check for and correct any other errors reported prior to the severe error. • If the message indicates an internal compile-time error, report the message to your IBM service representative.
U	Unrecoverable error	fatal error	<p>The compiler halts. An internal compile-time error has occurred. Report the message to your IBM service representative.</p>

Related information

- [“-qhalt” on page 71](#)
- [“Listings, messages, and compiler information” on page 23](#)

Return codes

For every compilation job or job step, the compiler generates a return code that indicates to the operating system the degree of success or failure it achieved:

Table 4. Return codes from compilation of a XL C/C++ V2.4.1 program

Return Code	Type of Error Detected	Compilation Result
0	No error detected; informational messages may have been issued.	Compilation completed. Successful execution anticipated.
4	Warning error detected.	Compilation completed. Execution may not be successful.
8	Error detected.	Compilation may have been completed. Successful execution not possible.
12	Severe error detected.	Compilation may have been completed. Successful execution not possible.
16	Terminating error detected.	Compilation terminated abnormally. Successful execution not possible.
33	A library level prior to the z/OS z/OS V2.4 Language Environment® library level was used.	Compilation terminated abnormally. Successful execution not possible.

The return code indicates the highest possible error severity that the compiler detected. Therefore, a particular entry in the "Type of Error Detected" column, includes *all* error types above it. For example,

return code 12 indicates that the compiler has issued a severe error and may have also issued any combination of error, warning, and informational messages. But it does not necessarily mean that all these error types are present in that particular compile.

Compiler listings

A listing is a compiler output file (with a `.lst` suffix) that contains information about a particular compilation. As a debugging aid, a compiler listing is useful for determining what has gone wrong in a compilation.

To produce a listing, you can compile with `-qlist`, which provides different types of information:

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

Heading information

The first page of the listing is identified by the product number, the compiler version and release numbers, the date and time compilation began (formatted according to the current locale), and the page number.

Pseudo Assembly Listing

The `-qlist` compiler option generates a listing of the machine instructions in the object module in a form similar to assembler language. This Pseudo Assembly listing displays the source statement line numbers and the line number of any inlined code to aid you in debugging inlined code.

Related information

- [“Listings, messages, and compiler information” on page 23](#)

Chapter 2. Configuring compiler defaults

When you compile an application with XL C/C++ V2.4.1, the compiler uses default settings that are determined in a number of ways:

- Internally defined settings. These settings are predefined by the compiler and you cannot change them.
- Settings defined by system environment variables. Certain environment variables are required by the compiler; others are optional. You might have already set some of the basic environment variables during the installation process. [“Setting environment variables” on page 11](#) provides a complete list of the required and optional environment variables you can set or reset after installing the compiler.
- Settings defined in the compiler configuration file, `/usr/lpp/cbclib/xlclang/etc/xlclang.cfg`. The compiler requires many settings that are determined by its configuration file. The configuration file is automatically generated during the installation procedure. However, you can customize this file after installation, to specify additional compiler options, default option settings, library search paths, and other settings. Information on customizing the configuration file is provided in [“Using custom compiler configuration files” on page 11](#).

Setting environment variables

The following sections discuss the environment variables you can set for XL C/C++ V2.4.1 and applications you have compiled with it:

- [“Compile-time and link-time environment variables” on page 11](#)
- [“Runtime environment variables” on page 11](#)

Compile-time and link-time environment variables

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

LANG

Specifies the locale for your operating system. The default locale used by the compiler for messages and help files is United States English, `en_US`. For more information on setting the `LANG` environment variable to use an alternate locale, see your operating system documentation.

NLSPATH

Specifies the directory search path for finding the compiler message and help files.

CLC_CONFIG

Specifies the location of a custom configuration file to be used by the compiler. The file name must be given with its absolute path. For more information, see [“Using custom compiler configuration files” on page 11](#).

Runtime environment variables

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

LIBPATH

Allows an absolute or relative pathname to be searched when loading a DLL.

Using custom compiler configuration files

The XL C/C++ V2.4.1 compiler contains a sample configuration file `/usr/lpp/cbclib/xlclang/sample/xlclang.cfg`, which can be used to customize the default configuration file `/usr/lpp/cbclib/`

`xlclang/etc/xlclang.cfg`. The configuration file specifies information that the compiler uses when you invoke it.

A configuration file is a UNIX file consisting of named sections called stanzas. Each stanza contains keywords called configuration file attributes, which are assigned values. The attributes are separated from their assigned value by an equal sign. A stanza can point to a default stanza by specifying the `use` keyword. This allows specifying common attributes in a default stanza and only the deltas in a specific stanza, referred to as the local stanza.

For any of the supported attributes not found in the configuration file, the **xlclang** utility uses the built-in defaults. It uses the first occurrence in the configuration file of a stanza or attribute it is looking for. Unsupported attributes, and duplicate stanzas and attributes are not diagnosed.

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

If you want users to be able to choose among several sets of compiler options, you might want to use custom configuration files for specific needs. For example, you might want to enable **-qlist** by default for compilations using the **xlclang** compiler invocation command. This is to avoid forcing your users to specify this option on the command line for every compilation, because **-qno1ist** is automatically in effect every time the compiler is called with the **xlclang** command.

You have several options for customizing configuration files:

- You can directly edit the default configuration file. In this case, the customized options will apply for all users for all compilations. The disadvantage of this option is that you will need to reapply your customizations to the new default configuration file that is provided every time you install a compiler update.
- You can use the default configuration file as the basis of customized copies that you specify at compile time with the `-F` option. In this case, the custom file overrides the default file on a per-compilation basis.
- You can create custom, or user-defined, configuration files that are specified at compile time with the `CLC_CONFIG` environment variable. It can be specified on a per-compilation or global basis. Procedures for creating custom, user-defined configuration files are provided below.

Notes:

1. The difference between specifying values in the stanza and relying on the defaults provided by the **xlclang** utility is that the defaults provided by the **xlclang** utility will not override pragmas.
2. Any entry in the configuration file must occur on a single line. You cannot continue an entry over multiple lines.
3. After you apply service to the compiler, any customization to the configuration file might need to be adjusted.

Related reference

[“-F” on page 33](#)

Related information

[“Compile-time and link-time environment variables” on page 11](#)

Creating custom configuration files

The [default configuration file](#) is installed in `/usr/lpp/cbclib/xlclang/etc/xlclang.cfg`.

You can copy this file and make changes to the copy to support specific compilation requirements or to support other C or C++ compilation environments. The `-F` option is used to specify a configuration file other than the default. For example, to make **-qno1ist** the default for the **xlclang** compiler invocation command, add **-qno1ist** to the **clang** 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.

Only one stanza, in addition to the one referenced by the use attribute, is processed for any one invocation of the **xlclang** utility. By default, the stanza that matches the command name used to invoke the **xlclang** utility is used, but it can be overridden using the **-F** flag option as described in the example below.

Example: You can use the **-F** option with the compiler invocation command to make links to select additional stanzas or to specify a stanza or another configuration file:

```
xlclang++ myfile.C -Fmyconfig:SPECIAL
```

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

Example of default configuration file

The default configuration file is installed in `/usr/lpp/cbclib/xlclang/etc/xlclang.cfg`. The content is as below:

```
*
* FUNCTION: z/OS V2.4.1 XL C/C++ Compiler Configuration file
*
* Licensed Materials - Property of IBM
* 5650-ZOS Copyright IBM Corp. 2019.
* US Government Users Restricted Rights - Use, duplication or
* disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
*

* Clang C compiler
clang:      use              = DEFLT

* Clang C++ compiler
clang++:    use              = DEFLT
           options          = -D_XOPEN_SOURCE=600,-D__static_assert=static_assert,-Wno-
parentheses,-Wno-unused-value

* common definitions
DEFLT:      cppcomp          = /usr/lpp/cbclib/xlclang/exe/clcdrvr
           ccomp             = /usr/lpp/cbclib/xlclang/exe/clcdrvr
           as                 = /bin/c89
           ld_c               = /bin/c89
           ld_cpp             = /bin/cxx
           xlc                = /usr/lpp/cbclib/xlclang/bin/xlclang
           xlCcopt            = -D_XOPEN_SOURCE
           sysobj             = cee.sceobj:cee.sceecpp
           syslib_x           = cee.sceebnd2:sys1.csslib
           exportlist_c_x     = cee.sceelib(celhs003,celhs001)
           exportlist_cpp_x   = cee.sceelib(celhs003,celhs001,celhscpp)
           exportlist_c_64    = cee.sceelib(celqs003)
           exportlist_cpp_64  = cee.sceelib(celqs003,celqscpp,cxxrt64)
           cinc               = -isystem/usr/include/le
           cppinc              = -isystem/usr/include/c++
           options            = -D_UNIX03_WITHDRAWN,-L/usr/lpp/cbclib/lib
           libraries          = -libmcmp
           steplib             = cbc.sclccmp
```

Examples of stanzas in custom configuration files

```
clang++: use=DEFLT
        options+=-qlist
```

This example specifies that **-qlist** is to be used for any compilation called by the `xlclang++` command.

```
DEFLT: options = -L/home/user/lib
        libraries = -lmylib
```

This example specifies that all compilations should link with `/home/user/lib/libmylib.a`.

Editing the default configuration file

The configuration file specifies information that the compiler uses when you invoke it. XL C/C++ V2.4.1 provides the default configuration file `/usr/lpp/cbclib/xlclang/etc/xlclang.cfg` at installation time.

Configuration file attributes

A stanza in the configuration file can contain the following attributes:

acceptable_rc

Enables you to specify a number that represents a return code value for a program invoked by the **xlclang** utility. The **xlclang** utility does not place any restriction on the value assigned to the `acceptable_rc` attribute. `acceptable_rc` can appear as part of any stanza in the configuration file.

Note: If the `acceptable_rc` attribute is not specified in the configuration file, the **xlclang** utility will assign the value from a `c89 prefix_ACCEPTABLE_RC` environment variable, if it is exported, to the `acceptable_rc`, otherwise it will default to 4. The command name used to invoke the **xlclang** utility determines the prefix that the **xlclang** utility will use when looking for a `prefix_ACCEPTABLE_RC` environment variable. For example, if the **xlclang** utility is invoked using the **xlclang++** command name, the **xlclang** utility will look for `_CXX_ACCEPTABLE_RC` and, if found, use it. If the `acceptable_rc` attribute is specified in the configuration file, the **xlclang** utility will use the value specified in the configuration file and will ignore an exported `prefix_ACCEPTABLE_RC` environment variable.

as

Path name to be used for the assembler. The default is `/bin/c89`.

asmlib

Specifies assembler macro libraries to be used when assembling the assembler source code.

asopt

The list of options for the assembler and not for the compiler. These override all normal processing by the compiler and are directed to the assembler specified in the `as` attribute. Options are specified following the **c89** utility syntax.

asuffix

The suffix for archive files. The default is `a`.

asuffix_host

The suffix for archive data sets. The default is `LIB`.

ccomp

The C compiler. The default is `/usr/lpp/cbclib/xlclang/exe/clcdrvr`.

cinc

A comma separated list of directories used to search for C header files. The default for this attribute is `-isystem/usr/include/le`.

cppcomp

The C++ compiler. The default is `/usr/lpp/cbclib/xlclang/exe/clcdrvr`.

cppinc

A comma separated list of directories used to search for C++ header files. The default for this attribute is `-isystem/usr/include/c++`, `-isystem/usr/include/le`. For further information on the list of search places used by the compiler to search for system header files, see the note at the end of this list of configuration file attributes.

csuffix

The suffix for C source programs. The default is `c` (lowercase c).

cversion

The compiler version.

cxxsuffix

The suffix for C++ source files. The default is C (uppercase C).

exportlist_c_x

A colon separated list of data sets with member names indicating definition side-decks to be used to resolve symbols during the link-editing phase of XPLINK C applications. The default for this attribute is:

```
CEE.SCEELIB(CELHS003,CELHS001)
```

exportlist_cpp_x

A colon separated list of data sets with member names indicating definition side-decks to be used to resolve symbols during the link-editing phase of XPLINK C++ applications. The default for this attribute is:

```
CEE.SCEELIB(CELHS003,CELHSCPP,CELHS001)
```

exportlist_c_64

A colon separated list of data sets with member names indicating definition side-decks to be used to resolve symbols during the link-editing phase of 64-bit C applications. The default for this attribute is:

```
CEE.SCEELIB(CELQS003)
```

exportlist_cpp_64

A colon separated list of data sets with member names indicating definition side-decks to be used to resolve symbols during the link-editing phase of 64-bit C++ applications. The default for this attribute is:

```
CEE.SCEELIB(CELQS003,CELQSCPP,CXXRT64)
```

isuffix

The suffix for C preprocessed files. The default is i.

ixxsuffix

The suffix for C++ preprocessed files. The default is i.

ld

The path name to be used for the binder. The default is /bin/c89.

ld_c

The path name to be used for the binder when only C sources appear on the command line invoked with a C stanza. The default is /bin/c89.

ld_cpp

The path name to be used for the binder when at least one C++ source appears on the command line, or when a C++ stanza is used. The default is /bin/cxx.

libraries

libraries specifies the default libraries that the binder is to use at bind time. The libraries are specified using the **-llibname** syntax, with multiple library specifications separated by commas. The default is -L/usr/lpp/cbclib/lib,-libmcmp.

libraries2

libraries2 specifies additional libraries that the binder is to use at bind time. The libraries are specified using the **-llibname** syntax, with multiple library specifications separated by commas. The default is empty.

options

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

osuffix

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

pversion

The runtime library version.

ssuffix

The suffix for assembler files. The default is .s.

ssuffix_host

The suffix for assembler data sets. The default is ASM.

steplib

A colon separated list of data sets or keyword NONE used to set the STEPLIB environment variable. The default is NONE, which causes all programs to be loaded from LPA or linklist.

use

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

xlC

The path name of the C++ compiler invocation command. The default is /usr/lpp/cbc/lib/xlclang/bin/xlclang.

xlCcopt

A string of option flags, separated by commas, to be processed when the **xlclang++** command is used for compiling a C file.

xsuffix

The suffix for definition side-deck files. The default is x.

xsuffix_host

The suffix for definition side-deck data sets. The default is EXP.

Chapter 3. Compiler options reference

This section contains a summary of the compiler options available in XL C/C++ V2.4.1 by functional category, followed by detailed descriptions of the individual options. It also provides a list of supported GCC options.

Related information

- [“Specifying compiler options” on page 3](#)

Summary of compiler options by functional category

The XL C/C++ V2.4.1 options are grouped into the following categories. If the option supports an equivalent pragma directive, this is indicated. To get detailed information on any option listed, see the full description for that option.

- [“Output control” on page 17](#)
- [“Input control” on page 18](#)
- [“Language element control” on page 18](#)
- [“Template control \(C++ only\)” on page 19](#)
- [“Floating-point and integer control” on page 20](#)
- [“Error checking and debugging” on page 20](#)
- [“Listings, messages, and compiler information” on page 23](#)
- [“Optimization and tuning” on page 24](#)
- [“Object code control” on page 20](#)
- [“Linking” on page 25](#)
- [“Compiler customization” on page 25](#)

Output control

The options in this category control the type of output file the compiler produces, as well as the locations of the output. These are the basic options that determine the following aspects:

- The compiler components that will be invoked
- The preprocessing, compilation, and linking steps that will (or will not) be taken
- The kind of output to be generated

Table 5. Compiler output options	
Option name	Description
“-c” on page 29	Instructs the compiler to compile or assemble the source files only but do not link. With this option, the output is a .o file for each source file.
“-C” on page 28	When used in conjunction with the -E or -P options, preserves or removes comments in preprocessed output.
“-E” on page 31	Preprocesses the source files named in the compiler invocation, without compiling. The preprocessed file is output to the standard out.

Table 5. Compiler output options (continued)	
Option name	Description
“-o” on page 46	Specifies a name for the output object, assembler, executable, or preprocessed file.
“-P” on page 47	Preprocesses the source files named in the compiler invocation, without compiling, and creates an output preprocessed file for each input file.
“-qmakedep” on page 79	Produces the dependency files that are used by the make tool for each source file.
“-qshowmacros” on page 89	Emits macro definitions to preprocessed output.
“-W” on page 51	-W <i>option</i> passes the listed option directly to the preprocessor.

The following options are supported by XL C/C++ V2.4.1 for compatibility with other compilers. For details about these options, see [GNU Compiler Collection online documentation](http://gcc.gnu.org/onlinedocs/)(<http://gcc.gnu.org/onlinedocs/>).

- -M
- -MD
- -MF *file*
- -MG
- -MM
- -MMD
- -MP
- -MQ *target*
- -MT *target*





Input control

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

Table 6. Compiler input options	
Option name	Description
“-I” on page 34	Adds a directory to the search path for include files.
“-qasmlib” on page 57	Specifies assembler macro libraries to be used when assembling the assembler source code.
“-qinclude” on page 73	Specifies additional header files to be included in a compilation unit, as though the files were named in an <code>#include</code> statement in the source file.

Language element control

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

Table 7. Language element control options	
Option name	Description
“-D” on page 30	Defines a macro as in a <code>#define</code> preprocessor directive.
“-qasm (-fasm)” on page 56	Controls the interpretation and subsequent generation of code for assembler language extensions.
“-qdigraph” on page 67	Enables recognition of digraph key combinations  and operator keywords  to represent characters that are not found on some keyboards. Digraph key combinations include <code><:</code> , <code><%</code> , and so on.  Operator keywords include <code>and</code> , <code>or</code> , and so on. 
“-qlanglvl (-std)” on page 75	Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.
“-U” on page 49	Undefines a macro defined by the compiler or by the -D compiler option.
“-W” on page 51	-Wa, option passes the listed option directly to the assembler.

The following options are supported by XL C/C++ V2.4.1 for compatibility with other compilers. For details about these options, see [GNU Compiler Collection online documentation](http://gcc.gnu.org/onlinedocs/)(<http://gcc.gnu.org/onlinedocs/>).

- **-fconstexpr-depth**
- **-fexec-charset**
- **-ffreestanding**
- **-fgnu89-inline**
- **-fgnu-keywords**
- **-fhosted**
- **-foperator-names**
- **-frtti** (synonym for **-qrtti**)
- **-fsigned-char** (synonym for **-qchars=signed**)
- **-ftemplate-backtrace-limit**
- **-funsigned-char** (synonym for **-qchars=unsigned**)

Template control (C++ only)

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

Table 8. C++ template options	
Option name	Description
“-qtemplatedepth (-ftemplate-depth) (C++ only)” on page 95	Specifies the maximum number of recursively instantiated template specializations that will be processed by the compiler.

Floating-point and integer control

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

Table 9. Floating-point and integer control options	
Option name	Description
“-qchars (-fsigned-char, -funsigned-char)” on page 59	Determines whether all variables of type char is treated as signed or unsigned.
“-qfloat” on page 69	Selects different strategies for speeding up or improving the accuracy of floating-point calculations.

Object code control

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

Table 10. Object code control options	
Option name	Description
“-qrtti (-frtti) (C++ only)” on page 87	Generates runtime type identification (RTTI) information for classes with virtual functions.
“-qcompress” on page 60	Suppresses the generation of function names in the function control block, thereby reducing the size of your application's load module.
“-qeh (C++ only)” on page 67	Controls whether exception handling is enabled in the module being compiled.
“-qexportall” on page 68	Exports all externally defined functions and variables in the compilation unit so that a DLL application can use them.
“-r” on page 48	Produces a nonexecutable output file to use as an input file in another binder command call. This file may also contain unresolved symbols.
“-qcsect” on page 61	Instructs the compiler to generate CSECT names in the output object module.
“-qro” on page 85	Specifies the storage type for string literals.
“-qroconst” on page 85	Specifies the storage location for constant values.
“-s” on page 49	Strips the symbol table, line number information, and relocation information from the output file.

Error checking and debugging

The options in this category allow you to detect and correct problems in your source code. In some cases, these options can alter your object code, increase your compile time, or introduce runtime checking that

can slow down the execution of your application. The option descriptions indicate how extra checking can impact performance.

To control the amount and type of information you receive regarding the behavior and performance of your application, consult the options in [“Listings, messages, and compiler information”](#) on page 23.

Table 11. Error checking and debugging options	
Option name	Description
“-# (pound sign)” on page 27	Previews the compilation steps specified on the command line, without actually invoking any compiler components.
-fstack-protector (-qstackprotect)	Provides protection against malicious input data or programming errors that overwrite or corrupt the stack.
“-qsyntaxonly (-fsyntax-only)” on page 94	Performs syntax checking without generating an object file.
“-g” on page 34	Generates debugging information for use by a symbolic debugger, and makes the program state available to the debugging session at selected source locations.
“-qdebug” on page 63	Instructs the compiler to generate debugging information.
“-qgonumber” on page 70	Generates line number tables that correspond to the input source file for Debug Tool and CEEDUMP processing.
“-qrtcheck” on page 86	Generates compare-and-trap instructions which perform certain types of runtime checking. The messages can help you to debug your C and C++ programs.
“-qservice” on page 88	Places a <i>string</i> in the object module, which is displayed in the traceback if the application fails abnormally.

Options to control diagnostic messages formatting

The following options are supported by XL C/C++ V2.4.1 for compatibility with other compilers. For details about these options, see [GNU Compiler Collection online documentation](http://gcc.gnu.org/onlinedocs/)(<http://gcc.gnu.org/onlinedocs/>).

- **-fdiagnostics-show-option**
- **-felide-type**
- **-fshow-column**
- **-fshow-source-location**
- **-pedantic**

Options to request or suppress warnings

The following options are supported by XL C/C++ V2.4.1 for compatibility with other compilers. see [GNU Compiler Collection online documentation](http://gcc.gnu.org/onlinedocs/)(<http://gcc.gnu.org/onlinedocs/>).

- **-fsyntax-only** (synonym for **-qsyntaxonly**)

- -w
- -Wall
- -Wbad-function-cast
- -Wcast-align
- -Wchar-subscripts
- -Wcomment
- -Wconversion
- -Wc++11-compat
- -Wdelete-non-virtual-dtor
- -Wempty-body
- -Wenum-compare
- -Werror=foo
- -Weverything
- -Wfatal-errors
- -Wfloat-equal
- -Wfoo
- -Wformat
- -Wformat=n
- -Wformat=2
- -Wformat-nonliteral
- -Wformat-security
- -Wformat-y2k
- -Wignored-qualifiers
- -Wimplicit-int
- -Wimplicit-function-declaration
- -Wimplicit
- -Wmain
- -Wmissing-braces
- -Wmissing-field-initializers
- -Wmissing-prototypes
- -Wnarrowing
- -Wno-attributes
- -Wno-builtin-macro-redefined
- -Wno-deprecated
- -Wno-deprecated-declarations
- -Wno-division-by-zero
- -Wno-endif-labels
- -Wno-format
- -Wno-format-extra-args
- -Wno-format-zero-length
- -Wno-int-conversion
- -Wno-invalid-offsetof
- -Wno-int-to-pointer-cast

- -Wno-multichar
- -Wnonnull
- -Wno-return-local-addr
- -Wno-unused-result
- -Wno-virtual-move-assign
- -Wnon-virtual-dtor
- -Woverlength-strings
- -Woverloaded-virtual
- -Wpadded
- -Wparantheses
- -Wpointer-arith
- -Wpointer-sign
- -Wreorder
- -Wreturn-type
- -Wsequence-point
- -Wshadow
- -Wsign-compare
- -Wsign-conversion
- -Wsizeof-pointer-memaccess
- -Wstack-protector
- -Wswitch
- -Wsystem-headers
- -Wtautological-compare
- -Wtype-limits
- -Wtrigraphs
- -Wundef
- -Wuninitialized
- -Wunknown-pragmas
- -Wunused
- -Wunused-label
- -Wunused-parameter
- -Wunused-variable
- -Wunused-value
- -Wvariadic-macros
- -Wvarargs
- -Wvla
- -Wwrite-strings

Listings, messages, and compiler information

The options in this category allow your control over the listing file, as well as how and when to display compiler messages. You can use these options in conjunction with those described in [“Error checking and debugging”](#) on page 20 to provide a more robust overview of your application when checking for errors and unexpected behavior.

<i>Table 12. Listings and messages options</i>	
Option name	Description
“-qlist” on page 78	Produces a compiler listing file that includes object and constant area sections.
“-qoffset” on page 82	Lists offset addresses relative to entry points of functions.
“-qphaseid, -qphsinfo” on page 83	Causes each compiler component (phase) to issue an informational message as each phase begins execution and reports the time taken in each compilation phase.

Optimization and tuning

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

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

<i>Table 13. Optimization and tuning options</i>	
Option name	Description
“-qansialias” on page 52	Indicates to the compiler that the code strictly follows the type-based aliasing rule in the ISO C and C++ standards, and can therefore be compiled with higher performance optimization of the generated code.
“-qassert” on page 58	Enables optimizations for restrict qualified pointers.
“-qarch” on page 54	Specifies the processor architecture for which the code (instructions) should be generated.
“-qcompact” on page 60	Avoids optimizations that increase code size.
“-qignerrno” on page 72	Allows the compiler to perform optimizations as if system calls would not modify <code>errno</code> .
“-qinline” on page 73	Attempts to inline functions instead of generating calls to those functions, for improved performance.
“-qlibansi” on page 77	Assumes that all functions with the name of an ANSI C library function are in fact the system functions.
“-qmaxmem” on page 80	Limits the amount of memory that the compiler allocates while performing specific, memory-intensive optimizations to the specified number of kilobytes.
“-O, -qoptimize” on page 44	Specifies whether to optimize code during compilation and, if so, at which level.

Table 13. Optimization and tuning options (continued)	
Option name	Description
“-qstrict” on page 92	Ensures that optimizations that are done by default at the -O3 and higher optimization levels, and, optionally at -O2 , do not alter the semantics of a program.
“-qstrict_induction” on page 93	Prevents the compiler from performing induction (loop counter) variable optimizations. Such optimizations might be problematic when integer overflow operations involving the induction variables occurs.
“-qthreaded” on page 96	Indicates to the compiler whether it must generate threadsafe code.
“-qtune” on page 96	Tunes instruction selection, scheduling, and other architecture-dependent performance enhancements to run best on a specific hardware architecture.

Linking

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

Table 14. Linking options	
Option name	Description
“-e” on page 32	Specifies an entry point for a shared object.
“-L” on page 35	At link time, searches the directory path for library files specified by the -l option.
“-l” on page 36	Searches for the specified library file <i>libkey.a</i> .
“-W” on page 51	-Wl,option passes the listed option directly to the binder.

Portability and migration

The options in this category can help you port software from other platforms to z/OS.

Table 15. Portability and migration options	
Option name	Description
“-qascii” on page 55	Enables your application to process ASCII data natively at execution time.

Compiler customization

The options in this category allow you to specify alternative locations for compiler components, configuration files, standard include directories, and internal compiler operation. These options are useful for specialized installations, testing scenarios, and the specification of additional command-line options.

Table 16. Compiler customization options	
Option name	Description
“-qoptfile” on page 83	Specifies an options file that contains a list of additional command line options to be used for the compilation.
“-F” on page 33	Names an alternative configuration file or stanza for the compiler.
“-qmemory” on page 81	Improves compile-time performance by using a memory file in place of a temporary work file, if possible.
“-qspill” on page 90	Specifies the size (in bytes) of the register spill space, the internal program storage areas used by the optimizer for register spills to storage.
“-W” on page 51	Passes one or more options to a component that is indicated by a single lower case letter following -W .

Individual option descriptions

This section contains descriptions of the individual compiler options available in XL C/C++ V2.4.1.

For each option, the following information is provided:

Category

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

Pragma equivalent

Many compiler options allow you to use an equivalent pragma directive to apply the option's functionality within the source code, limiting the scope of the option's application to a single source file, or even selected sections of code.

When an option supports the **#pragma name** form of the directive, this is indicated.

Purpose

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

Syntax

This section provides the syntax for the option, and where an equivalent **#pragma name** is supported, the specific syntax for the pragma.

Note that you can also use the C99-style `_Pragma` operator form of any pragma; although this syntax is not provided in the option descriptions. For complete details on pragma syntax, see [“Pragma directive syntax” on page 101](#)

Defaults

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

Parameters

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

Usage

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

Predefined macros

Many compiler options set macros that are protected (that is, cannot be undefined or redefined by the user). Where applicable, any macros that are predefined by the option, and the values to which they are defined, are listed in this section. A reference list of these macros (as well as others that are defined independently of option setting) is provided in [Chapter 5, “Compiler predefined macros,” on page 115](#)

Examples

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

-# (pound sign)

Category

[Error checking and debugging](#)

Pragma equivalent

None.

Purpose

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

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

Syntax

►► -# ◀◀

Usage

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

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

Predefined macros

None.

Examples

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

```
xlclang myprogram.c -#
```

Related information

- [“-v, -V” on page 50](#)

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

Category

Input control

Pragma equivalent

None.

Purpose

Compiles any file as a C++ language file.

Syntax

► -+ ◀

Usage

You can use **-+** to compile a file with any suffix other than `.a`, `.o`, `.so`, `.S`, or `.s`. If you do not use the **-+** option, files must have a suffix of `.C` (uppercase C), `.cc`, `.cpp`, or `.cxx` 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.

Predefined macros

None.

Examples

To compile the file `myprogram.cplsp1s` as a C++ source file, enter:

```
xlclang -+ myprogram.cplsp1s
```

-C

Category

Output control

Pragma equivalent

None.

Purpose

When used in conjunction with the **-E** or **-P** options, preserves or removes comments in preprocessed output.

When **-C** is in effect, comments are preserved.

Syntax

► -C ◀

Defaults

None.

Usage

The **-C** option has no effect without either the **-E** or the **-P** option. If **-E** is specified, continuation sequences are preserved in the output. If **-P** is specified, continuation sequences are stripped from the output, forming concatenated output lines.

Predefined macros

None.

Examples

To compile `myprogram.c` to produce a file `myprogram.i` that contains the preprocessed program text including comments, enter:

```
xlclang myprogram.c -P -C
```

Related information

- [“-E” on page 31](#)
- [“-P” on page 47](#)

-c

Category

[Output control](#)

Pragma equivalent

None.

Purpose

Instructs the compiler to compile or assemble the source files only but do not link. With this option, the output is a `.o` file for each source file.

Syntax

►► -c ►►

Defaults

By default, the compiler invokes the binder to link object files into a final executable.

Usage

When this option is in effect, the compiler creates an output object file, *file_name.o*, for each valid source file, such as *file_name.c*, *file_name.i*, *file_name.C*, *file_name.cpp*, or *file_name.s*. You can use the **-o** option to provide an explicit name for the object file.

The **-c** option is overridden if the **-E**, **-P**, or **-fsyntax-only (-qsyntaxonly)** option is specified.

Predefined macros

None.

Examples

To compile `myprogram.c` to produce an object file `myprogram.o`, but no executable file, enter the command:

```
xlclang myprogram.c -c
```

To compile `myprogram.c` to produce the object file `new.o` and no executable file, enter the command:

```
xlclang myprogram.c -c -o new.o
```

Related information

- [“-E” on page 31](#)
- [“-o” on page 46](#)
- [“-P” on page 47](#)
- [“-qsyntaxonly \(-fsyntax-only\)” on page 94](#)

-D

Category

[Language element control](#)

Pragma equivalent

None.

Purpose

Defines a macro as in a `#define` preprocessor directive.

Syntax

➤ `-D` — *name* — `=` — *definition* ➤

Defaults

Not applicable.

Parameters

name

The macro you want to define. **-Dname** is equivalent to `#define name`. For example, **-DCOUNT** is equivalent to `#define COUNT`.

definition

The value to be assigned to *name*. **-Dname=definition** is equivalent to `#define name definition`. For example, **-DCOUNT=100** is equivalent to `#define COUNT 100`.

Usage

Using the `#define` directive to define a macro name already defined by the **-D** option will result in an error condition.

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

Predefined macros

The compiler configuration file uses the **-D** option to predefine several macro names for specific invocation commands. For details, see the configuration file for your system.

Examples

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

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

Related information

- [“-U” on page 49](#)
- [Chapter 5, “Compiler predefined macros,” on page 115](#)

-E

Category

[Output control](#)

Pragma equivalent

None.

Purpose

Preprocesses the source files named in the compiler invocation, without compiling. The preprocessed file is output to the standard out.

Syntax

➤ -E ➤

Defaults

By default, source files are preprocessed, compiled, and linked to produce an executable file.

Usage

Source files with unrecognized file name suffixes are treated and preprocessed as C files.

Unless **-C** is specified, comments are replaced in the preprocessed output by a single space character. New lines and `#line` directives are issued for comments that span multiple source lines.

The **-E** option overrides the **-P** and **-fsyntax-only** (**-qsyntaxonly**) options. The combination of **-E -o** stores the preprocessed result in the file specified by **-o**.

Predefined macros

None.

Examples

To compile `myprogram.c` and send the preprocessed source to standard output, enter:

```
xlclang 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,b) ; /* Comment in a macro function argument*/
```

the output will be:

```
int a ;
int b ;
int c ;
c = a + b ;
```

Related information

- [“-C” on page 28](#)
- [“-P” on page 47](#)
- [“-qsyntaxonly \(-fsyntax-only\)” on page 94](#)

-e

Category

[Linking](#)

Pragma equivalent

None.

Purpose

Specifies the name of the function to be used as the entry point of the program.

Syntax

►► -e — *function* ◄◄

Defaults

The function `//ceestart` is the default.

Parameters

function

The name of the function to be used as the entry point of the program.

Usage

This can be useful when creating a fetchable program, or a non-C or non-C++ main, such as a COBOL program. Non-C++ linkage symbols of up to 1024 characters in length may be specified. You can specify an S-name by preceding the function name with double slash (/ /). An S-name is a short external symbol name, which is produced by the compiler when compiling C programs with the **NOLONGNAME** option.

Predefined macros

None.

-F

Category

[Compiler customization](#)

Pragma equivalent

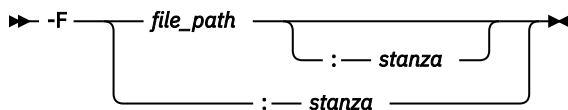
None.

Purpose

Names an alternative configuration file or stanza for the compiler.

Note: This option is not equivalent to the **-F** option that GCC provides.

Syntax



Defaults

By default, the compiler uses the configuration file that is configured at installation time, and uses the stanza defined in that file for the invocation command currently being used.

Parameters

file_path

The full path name of the alternate compiler configuration file to use.

stanza

The name of the configuration file stanza to use for compilation. This directs the compiler to use the entries under that *stanza* regardless of the invocation command being used.

Usage

Note that any file names or stanzas that you specify with the **-F** option override the defaults specified in the system configuration file. If you have specified a custom configuration file with the `CLC_CONFIG` environment variable, that file is processed before the one specified by the **-F** option.

Predefined macros

None.

Examples

To compile `myprogram.c` using a stanza called `debug` that you have added to the default configuration file, enter:

```
xlclang myprogram.c -F:debug
```

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

```
xlclang myprogram.c -F/usr/tmp/myconfig.cfg
```

To compile `myprogram.c` using the stanza debug you have created in a configuration file called `/usr/tmp/myconfig.cfg`, enter:

```
xlclang myprogram.f -F/usr/tmp/myconfig.cfg:debug
```

Related information

- [“Using custom compiler configuration files” on page 11](#)
- [“Specifying compiler options in a configuration file” on page 4](#)
- [“Compile-time and link-time environment variables” on page 11](#)

-g

Category

[Error checking and debugging](#)

Pragma equivalent

None.

Purpose

Generates debugging information for use by a symbolic debugger, and makes the program state available to the debugging session at selected source locations.

Program state refers to the values of user variables at certain points during the execution of a program.

When the **-O2** optimization level is in effect, the debug capability is completely supported.

When an optimization level higher than **-O2** is in effect, the debug capability is limited.

Syntax

► -g ◀

Defaults

Not applied.

Examples

Use the following command to compile `myprogram.c` and generate an executable program called `testing` for debugging:

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

Related information

- [“-O, -qoptimize” on page 44](#)

-I

Category

[Input control](#)

Pragma equivalent

None.

Purpose

Adds a directory to the search path for include files.

Syntax

► **-I** *directory_path* ◄

Defaults

See [“Directory search sequence for included files” on page 6](#) for a description of the default search paths.

Parameters

directory_path

The path for the directory where the compiler should search for the header files.

Usage

If the **-I** directory option is specified both in the configuration file and on the command line, the paths specified in the configuration file are searched first. The **-I** directory option can be specified more than once on the command line. If you specify more than one **-I** option, directories are searched in the order that they appear on the command line.

The **-I** option has no effect on files that are included using an absolute path name.

Predefined macros

None.

Examples

To compile `myprogram.c` and search `/usr/tmp` and then `/oldstuff/history` for included files, enter:

```
xlclang myprogram.c -I/usr/tmp -I/oldstuff/history
```

Related information

- [“-qinclude” on page 73](#)
- [“Directory search sequence for included files” on page 6](#)
- [“Specifying compiler options in a configuration file” on page 4](#)

-L

Category

[Linking](#)

Pragma equivalent

None.

Purpose

At link time, searches the directory path for library files specified by the **-l** option.

Syntax

►► -L — *directory_path* ►◄

Defaults

The default is to search only the standard directories. See the compiler configuration file for the directories that are set by default.

Parameters

directory_path

The path for the directory which should be searched for library files.

Usage

Paths specified with the **-L** compiler option are only searched at link time.

If the **-L*directory*** option is specified both in the configuration file and on the command line, search paths specified in the configuration file are the first to be searched at link time.

The **-L** compiler option is cumulative. Subsequent occurrences of **-L** on the command line do not replace, but add to, any directory paths specified by earlier occurrences of **-L**.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the directory `/usr/tmp/old` is searched for the library `libspfiles.a`, enter:

```
xlclang myprogram.c -lspfiles -L/usr/tmp/old
```

Related information

- [“-l” on page 36](#)

-l

Category

[Linking](#)

Pragma equivalent

None.

Purpose

Searches for the specified library file *libkey.a*.

Syntax

►► -l — *key* ►◄

Defaults

The compiler default is to search only some of the compiler runtime libraries. The default configuration file specifies the default library names to search for with the **-l** compiler option, and the default search path for libraries with the **-L** compiler option.

The C and C++ runtime libraries are automatically added.

Parameters

key

The name of the library minus the `lib` and `.a` or `.so` characters.

Usage

You must also provide additional search path information for libraries not located in the default search path. The search path can be modified with the **-L** option.

The **-l** option is cumulative. Subsequent appearances of the **-l** option on the command line do not replace, but add to, the list of libraries specified by earlier occurrences of **-l**. Libraries are searched in the order in which they appear on the command line, so the order in which you specify libraries can affect symbol resolution in your application.

Predefined macros

None.

Examples

To compile `myprogram.c` and link it with library `libmylibrary.so` or `libmylibrary.a` that is found in the `/usr/mylibdir` directory, enter the following command. Preference is given to `libmylibrary.so` over `libmylibrary.a`.

```
xlclang myprogram.c -lmylibrary -L/usr/mylibdir
```

Related information

- [“-L” on page 35](#)
- [“Specifying compiler options in a configuration file” on page 4](#)

-M

Category

[“Output control” on page 17](#)

Pragma equivalent

None.

Purpose

Instructs the compiler to generate a dependency file or dependency files that can be used by the **make** utility.

Syntax

► **-M** ◀

Defaults

The default is to search only the standard directories. See the compiler configuration file for the directories that are set by default.

Usage

The compiler will generate as many dependency files as there are source files specified. **-M** is the equivalent of specifying **-qmakep** with no suboption.

Dependency file name can be overridden by the **-MF** option.

Predefined macros

None.

Examples

To compile `myprogram.c` and create an output file named `myprogram.d`, enter:

```
xlclang -c -M myprogram.c
```

Related information

- [“-qmakep” on page 79](#)
- [“-MF” on page 39](#)

-MD

Category

[“Output control” on page 17](#)

Pragma equivalent

None.

Purpose

Instructs the compiler to generate a dependency output file as a side effect of the compilation process.

Syntax

►► -MD ►►

Defaults

None.

Usage

-MD is equivalent to **-M -MF file**, except that **-E** is not implied.

If **-o** is also specified, its argument is used as the file but with suffix of `.d`; otherwise the name of the input file is used, by removing directory components and replacing any suffix with a `.d` suffix.

If **-MD** is used with **-E**, the **-o** argument specifies the preprocessing output file; otherwise the **-o** argument specifies a target object file.

Predefined macros

None.

Related information

- [“-M” on page 37](#)
- [“-MF” on page 39](#)

-MF

Category

[“Output control” on page 17](#)

Pragma equivalent

None.

Purpose

If **-M** or **-qmakedep** is specified, instructs the compiler to override the default name of the dependency file.

Syntax

►► -MF — *file_name* ◄◄

Defaults

None.

Usage

file_name can be either a file name or a directory. By default, the dependency file name and path is the same as the **-o** compiler option but with `.d` suffix. If a directory is specified, the default dependency file name is used and placed in this directory. If a relative file name is specified, it is relative to the current working directory.

Notes:

1. The argument of *file_name* can not be the name of a data set.
2. If the file specified by **-MF** already exists, it will be overwritten. Moreover, if the output path specified does not exist or is write-protected, an error message will be issued.
3. If you specify a single file name for the **-MF** option when compiling multiple source files, each generated dependency file overwrites the previous one. Only a single output file will be generated for the last source file specified on the command line.

Predefined macros

None.

Examples

You can refer to the following table for detail usage of **-M** and **-MF**:

Table 17. Example of using -M and -MF

Description	Command	Dependency File
-MF is not specified	<code>xlclang -c -M t.c</code>	<code>./t.d</code> is generated.
	<code>xlclang -M -c -o obj.o t.c</code>	<code>./obj.d</code> is generated.
	<code>xlclang -c -M -o dir/ t.c</code>	<code>./dir/t.d</code> is generated if <code>./dir</code> is writable.
-MF specifies a file	<code>xlclang -c -qmakedep -MF dep.u t.c</code>	<code>./dep.d</code> is generated
	<code>xlclang -c -o obj.o -M -MF ../dep.x t.c</code>	<code>../dep.x</code> is generated
	<code>xlclang -c -M -MF dir/dep.d a.c b.c</code>	<code>./dir/dep.d</code> is generated for <code>b.c</code> only.
-MF specifies a directory	<code>xlclang -c -M -MF dir/ a.c b.c</code>	<code>./dir/a.d</code> and <code>./dir/b.d</code> are generated for <code>a.c</code> and <code>b.c</code> respectively if <code>./dir/</code> is writable.

Related information

- [“-M” on page 37](#)
- [“-qmakedep” on page 79](#)

-MG

Category

[“Output control” on page 17](#)

Pragma equivalent

None.

Purpose

If **-M** or **-qmakedep** is specified, this option instructs the compiler to include missing header files into the make dependencies file.

Syntax

➤ -MG ➤

Defaults

None.

Usage

When used with **-qmakedep=pponly**, **-MG** instructs the compiler to include missing header files into the make dependencies file and suppress diagnostic messages about missing header files.

When used with **-M**, **-qmakedep**, or **-qmakedep=gcc**, **-MG** instructs the C compiler to include missing header files into the make dependency output file, but the C compiler emits only warning messages and proceeds to create an object file if the missing headers do not cause subsequent severe compile errors.

Predefined macros

None.

Related information

- [“-M” on page 37](#)
- [“-qmakedep” on page 79](#)

-MM

Category

[“Output control” on page 17](#)

Pragma equivalent

None.

Purpose

Like **-M** but do not mention header files that are found in system header directories, nor header files that are included, directly or indirectly, from such a header.

Syntax

➤ **-MM** ➤

Defaults

None.

Usage

The choice of angle brackets or double quotes in an `#include` directive does not in itself determine whether that header appears in **-MM** dependency output.

Predefined macros

None.

Related information

- [“-M” on page 37](#)

-MMD

Category

[“Output control” on page 17](#)

Pragma equivalent

None.

Purpose

Like **-MD** except mention only user header files, not system header files.

Syntax

► **-MMD** ◄

Defaults

None.

Predefined macros

None.

Related information

- [“-MD” on page 38](#)

-MQ

Category

[“Output control” on page 17](#)

Pragma equivalent

None.

Purpose

If **-M** or **-qmakedep** is specified, this option sets the target to the *<target_name>* instead of the default target name.

Syntax

► **-MQ** — *target_name* ◄

Defaults

None.

Usage

The **-MQ** option is useful in cases where the target contains characters that have special meaning in make. See the following example:

```
> xlcclang -MQ '$(prefix)t.o' -qmakedep=gcc t.c
$$$(prefix)t.o : t.c \
    t1.h \
    t2.h
```

If the **-MQ** option is specified multiple times, the targets from each specification are included in the dependency file.

If **-MT** and **-MQ** are mixed on the command line, the targets from all **-MQ** flags will precede the targets from all **-MT** flags when they are emitted in the make dependency file.

Note: **-MQ** is the same as **-MT** except that **-MQ** escapes any characters that have special meaning in make.

Predefined macros

None.

Related information

- [“-M” on page 37](#)
- [“-MT” on page 43](#)
- [“-qmakedep” on page 79](#)

-MT

Category

[“Output control” on page 17](#)

Pragma equivalent

None.

Purpose

If **-M** or **-qmakedep** is specified, this option sets the target to the *<target_name>* instead of the default target name.

Syntax

➤ **-MT** — *target_name* ➤

Defaults

None.

Usage

This option is useful in cases where the target is not in the same directory as the source or when the same dependency rule applies to more than one target.

When **-MT** is used with **-M** or **-qmakedep** with no suboption, all targets are repeated for each dependency. See the following example:

```
> xlc -M -MT t1.o -MT t2.o t.c
t1.o t2.o : t.c
t1.o t2.o : t1.h
t1.o t2.o : t2.h
```

When **-MT** is used with **-qmakedep=gcc** or **-qmakedep=ppon1y**, all targets appear on a single line containing all dependencies. See the following example:

```
> xlcclang -M -MT t1.o -MT t2.o -qmakedep=gcc t.c
t1.o t2.o : t.c \
  t1.h \
  t2.h
```

If the **-MT** option is specified multiple times, the targets from each specification are included in the dependency file.

Predefined macros

None.

Related information

- [“-M” on page 37](#)
- [“-MQ” on page 42](#)
- [“-qmakedep” on page 79](#)

-O, -qoptimize

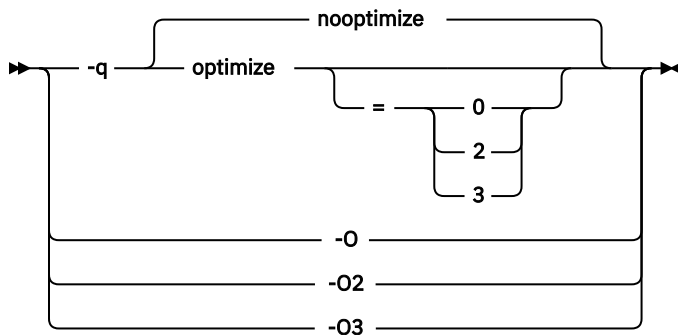
Category

[Optimization and tuning](#)

Purpose

Specifies whether to optimize code during compilation and, if so, at which level.

Syntax



Defaults

-qnooptimize

Parameters

nooptimize

Performs only quick local optimizations such as constant folding and elimination of local common subexpressions.

-O | -O2 | -qoptimize=2

Performs optimizations that the compiler developers considered the best combination for compilation speed and runtime performance. The optimizations may change from product release to release. If you need a specific level of optimization, specify the appropriate numeric value.

This setting implies **-qstrict** and **-qnostrict_induction**, unless explicitly negated by **-qstrict_induction** or **-qnostrict**.

-O3 | -qoptimize=3

Performs additional optimizations that are memory intensive, compile-time intensive, or both. They are recommended when the desire for runtime improvement outweighs the concern for minimizing compilation resources.

-O3 applies the **-O2** level of optimization, but with unbounded time and memory limits. **-O3** also performs higher and more aggressive optimizations that have the potential to slightly alter the semantics of your program. The compiler guards against these optimizations at **-O2**. The aggressive optimizations performed when you specify **-O3** are:

- Aggressive code motion, and scheduling on computations that have the potential to raise an exception, are allowed.

Loads and floating-point computations fall into this category. This optimization is aggressive because it may place such instructions onto execution paths where they *will* be executed when they *may* not have been according to the actual semantics of the program.

For example, a loop-invariant floating-point computation that is found on some, but not all, paths through a loop will not be moved at **-O2** because the computation may cause an exception. At **-O3**, the compiler will move it because it is not certain to cause an exception. The same is true for motion of loads. Although a load through a pointer is never moved, loads off the static or stack base register are considered movable at **-O3**. Loads in general are not considered to be absolutely safe at **-O2** because a program can contain a declaration of a static array `a` of 10 elements and load `a[600000000003]`, which could cause a segmentation violation.

The same concepts apply to scheduling.

Example:

In the following example, at **-O2**, the computation of `b+c` is not moved out of the loop for two reasons:

- It is considered dangerous because it is a floating-point operation
- It does not occur on every path through the loop

At **-O3**, the code is moved.

```
...
int i ;
float a[100], b, c ;
for (i = 0 ; i < 100 ; i++)
{
    if (a[i] < a[i+1])
        a[i] = b + c ;
}
...
```

- Both **-O2** and **-O3** conform to the following IEEE rules.

With **-O2** certain optimizations are not performed because they may produce an incorrect sign in cases with a zero result, and because they remove an arithmetic operation that may cause some type of floating-point exception.

For example, $X + 0.0$ is not folded to X because, under IEEE rules, $-0.0 + 0.0 = 0.0$, which is $-X$. In some other cases, some optimizations may perform optimizations that yield a zero result with the wrong sign. For example, $X - Y * Z$ may result in a -0.0 where the original computation would produce 0.0 .

In most cases the difference in the results is not important to an application and **-O3** allows these optimizations.

-qmaxmem=-1 is set by default with **-O3**, allowing the compiler to use as much memory as necessary when performing optimizations.

Built-in functions do not change `errno` at **-O3**.

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

You can use the **-qstrict** and **-qstrict_induction** compiler options to turn off effects of **-O3** that might change the semantics of a program. Specifying **-qstrict** together with **-O3** invokes all the optimizations performed at **-O2**. 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.

Usage

Increasing the level of optimization may or may not result in additional performance improvements, depending on whether additional analysis detects further opportunities for optimization.

Compilations with optimizations may require more time and machine resources than other compilations.

Optimization can cause statements to be moved or deleted, and generally should not be specified along with the **-g** flag for debugging programs. The debugging information produced may not be accurate.

You can use `#pragma option_override` to specify the optimization options on subprogram level, which overrides optimization options that are specified on the command line.

Predefined macros

- `__OPTIMIZE__` is predefined to 2 when **-O** | **-O2** is in effect; it is predefined to 3 when **-O3** is in effect. Otherwise, it is undefined.
- `__OPTIMIZE_SIZE__` is predefined to 1 when **-O** | **-O2** | **-O3** and **-qcompact** are in effect. Otherwise, it is undefined.

Examples

To compile and optimize `myprogram.c`, enter:

```
xlclang myprogram.c -O3
```

Related information

- [“-qcompact” on page 60](#)
- [“-g” on page 34](#)
- [“-qignerrno” on page 72](#)
- [“-qstrict” on page 92](#)

-O

Category

[Output control](#)

Pragma equivalent

None.

Purpose

Specifies a name for the output object, assembler, executable, or preprocessed file.

Syntax

►► -o — *path* ◀◀

Defaults

See [“Types of output files” on page 3](#) for the default file names and suffixes produced by different phases of compilation.

Parameters

path

When you are using the option to compile from source files, *path* can be the name of a file. *path* can be a relative or absolute path name. When you are using the option to link from object files, *path* must be a file name.

You cannot specify a file name with a C or C++ source file suffix (`.C`, `.c`, or `.cpp`), such as `myprog.c`; this results in an error and neither the compiler nor the binder is invoked.

Usage

If you use the **-c** option with **-o**, you can compile only one source file at a time. In this case, if more than one source file name is specified, the compiler issues a warning message and ignores **-o**.

The **-E**, **-P**, and **-fsyntax-only** (**-qsyntaxonly**) options override the **-o** option.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the resulting executable is called `myaccount`, enter:

```
xlclang myprogram.c -o myaccount
```

To compile `test.c` to an object file only and name the object file `new.o`, enter:

```
xlclang test.c -c -o new.o
```

Related information

- [“-c” on page 29](#)
- [“-E” on page 31](#)
- [“-P” on page 47](#)
- [“-qsyntaxonly \(-fsyntax-only\)” on page 94](#)

-P

Category

[Output control](#)

Pragma equivalent

None.

Purpose

Preprocesses the source files named in the compiler invocation, without compiling, and creates an output preprocessed file for each input file.

The preprocessed output file has the same name as the input file but with a `.i` suffix.

Syntax

► **-P** ◀

Defaults

By default, source files are preprocessed, compiled, and linked to produce an executable file.

Usage

Source files with unrecognized file name suffixes are preprocessed as C files except those with a `.i` suffix.

`#line` directives are not generated.

Line continuation sequences are removed and the source lines are concatenated.

The **-P** option retains all white space including line-feed characters, with the following exceptions:

- All comments are reduced to a single space (unless **-C** is specified).
- Line feeds at the end of preprocessing directives are not retained.
- White space surrounding arguments to function-style macros is not retained.

The **-P** option is overridden by the **-E** option. The **-P** option overrides the **-c**, **-o**, and **-fsyntax-only** (**-qsyntaxonly**) option.

Predefined macros

None.

Related information

- [“-C” on page 28](#)
- [“-E” on page 31](#)
- [“-qsyntaxonly \(-fsyntax-only\)” on page 94](#)

-r

Category

[Object code control](#)

Pragma equivalent

None.

Purpose

Produces a nonexecutable output file to use as an input file in another binder command call. This file may also contain unresolved symbols.

Syntax

►► -r ◀◀

Defaults

Not applicable.

Usage

A file produced with this flag is expected to be used as an input file in another compiler invocation or binder command call.

Predefined macros

None.

Examples

To compile `myprogram.c` and `myprog2.c` into a single object file `mytest.o`, enter:

```
xlclang myprogram.c myprog2.c -r -o mytest.o
```

-S

Category

[Object code control](#)

Pragma equivalent

None.

Purpose

Strips the symbol table, line number information, and relocation information from the output file.

This command is equivalent to the operating system **strip** command.

Syntax

►► -s ►►

Defaults

The symbol table, line number information, and relocation information are included in the output file.

Usage

Specifying **-s** saves space, but limits the usefulness of traditional debug programs when you are generating debugging information using options such as **-g**.

Predefined macros

None.

Related information

- [“-g” on page 34](#)

-U

Category

[Language element control](#)

Pragma equivalent

None.

Purpose

Undefines a macro defined by the compiler or by the **-D** compiler option.

Syntax

►► -U — *name* ►►

Defaults

Many macros are predefined by the compiler; see [Chapter 5, “Compiler predefined macros,” on page 115](#) for those that can be undefined (that is, are not *protected*). The compiler configuration file also uses the **-**

D option to predefine several macro names for specific invocation commands; see the configuration file for your system for more information.

Parameters

name

The macro you want to undefine.

Usage

The **-U** option is not equivalent to the `#undef` preprocessor directive. It cannot undefine names defined in the source by the `#define` preprocessor directive. It can only undefine names defined by the compiler or by the **-D** option.

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

Predefined macros

None.

Examples

Assume that your operating system defines the name `__unix`, but you do not want your compilation to enter code segments conditional on that name being defined, compile `myprogram.c` so that the definition of the name `__unix` is nullified by entering:

```
xlclang myprogram.c -U__unix
```

Related information

- [“-D” on page 30](#)

-v, -V

Category

[Listings, messages, and compiler information](#)

Pragma equivalent

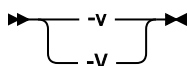
None.

Purpose

Reports the progress of compilation, by naming the programs being invoked and the options being specified to each program.

When the **-v** option is in effect, information is displayed in a comma-separated list. When the **-V** option is in effect, information is displayed in a space-separated list.

Syntax



Defaults

The compiler does not display the progress of the compilation.

Usage

The **-v** and **-V** options are overridden by the **-#** option.

Predefined macros

None.

Examples

To compile `myprogram.c` so you can watch the progress of the compilation and see messages that describe the progress of the compilation, the programs being invoked, and the options being specified, enter:

```
xlclang myprogram.c -v
```

Related information

- [“-# \(pound sign\)” on page 27](#)

-W

Category

[Compiler customization](#)

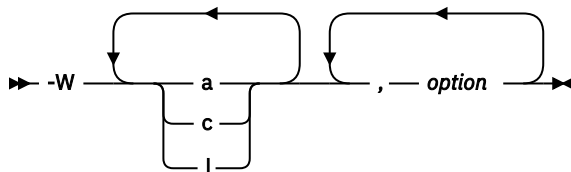
Pragma equivalent

None.

Purpose

Passes one or more options to a component that is indicated by a single lower case letter following **-W**.

Syntax



Parameters

a
Assembler

c
Compiler

l
Binder

option

Any option that is valid for the component to which it is being passed.

Usage

In the string following the **-W** option, use a comma as the separator for each option, and do not include any spaces. If you need to include a character that is special to the shell in the option string, precede the

character with a backslash. For example, if you use the option in the configuration file, you can use the escape sequence backslash comma (\,) to represent a comma in the parameter string.

You do not need the **-W** option to pass most options to the binder; unrecognized command-line options, except **-q** options, are passed to it automatically. Only binder options with the same letters as compiler options, such as **-v** or **-S**, strictly require **-W**.

Predefined macros

None.

Related information

- [“Invoking the compiler” on page 1](#)

-qansialias

Category

[Optimization and tuning](#)

Pragma equivalent

None.

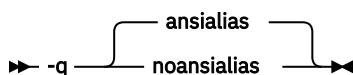
Purpose

Indicates to the compiler that the code strictly follows the type-based aliasing rule in the ISO C and C++ standards, and can therefore be compiled with higher performance optimization of the generated code.

When **-qansialias** is in effect, you are making a promise to the compiler that your source code obeys the constraints in the ISO standard. On the basis of using this compiler option, the compiler front end passes aliasing information to the optimizer, which performs optimization accordingly.

When **-qnoansialias** is in effect, the optimizer assumes that a given pointer of a given type can point to an external object or any object whose address is taken, regardless of type. This assumption creates a larger aliasing set at the expense of performance optimization.

Syntax



Defaults

-qansialias

Usage

When type-based aliasing is used during optimization, the optimizer assumes that pointers can only be used to access objects of the same type.

Type-based aliasing improves optimization in the following ways.

- It provides precise knowledge of what pointers can and cannot point at.
- It allows more loads to memory to be moved up and stores to memory moved down past each other, which allows the delays that normally occur in the original written sequence of statements to be overlapped with other tasks. These re-arrangements in the sequence of execution increase parallelism, which is desirable for optimization.

- It allows the removal of some loads and stores that otherwise might be needed in case those values were accessed by unknown pointers.
- It allows more identical calculations to be recognized ("commoning").
- It allows more calculations that do not depend on values modified in a loop to be moved out of the loop ("code motion").
- It allows better optimization of parameter usage in inlined functions.

Simplified, the rule is that you cannot safely dereference a pointer that has been cast to a type that is not closely related to the type of what it points at. The ISO C and C++ standards define the closely related types.

The following are not subject to type-based aliasing:

- Types that differ only in reference to whether they are signed or unsigned. For example, a pointer to a signed `int` can point to an unsigned `int`.
- Character pointer types (`char`, unsigned `char`, and in C but not C++ signed `char`).
- Types that differ only in their `const` or `volatile` qualification. For example, a pointer to a `const int` can point to an `int`.
- C++ types where one is a class derived from the other.

The XL C/C++ V2.4.1 compiler often exposes type-based aliasing violations that other compilers do not. The C++ compiler corrects most but not all suspicious and incorrect casts without warnings or informational messages.

In addition to the specific optimizations to the lines of source code that can be obtained by compiling with the **-qansialias** compiler option, other benefits and advantages, which are at the program level, are described below:

- It reduces the time and memory needed for the compiler to optimize programs.
- It allows a program with a few coding errors to compile with optimization, so that a relatively small percentage of incorrect code does not prevent the optimized compilation of an entire program.
- It positively affects the long-term maintainability of a program by supporting ISO-compliant code.

It is important to remember that even though a program compiles, its source code may not be completely correct. When you weigh tradeoffs in a project, the short-term expedience of getting a successful compilation by forgoing performance optimization should be considered with awareness that you may be nurturing an incorrect program. The performance penalties that exist today could worsen as the compilers that base their optimization on strict adherence to ISO rules evolve in their ability to handle increased parallelism.

The **-qansialias** compiler option only takes effect if the **-qoptimize** option is in effect.

Although type-based aliasing does not apply to the `volatile` and `const` qualifiers, these qualifiers are still subject to other semantic restrictions. For example, casting away a `const` qualifier might lead to an error at run time.

Predefined macros

None.

Examples

The following example executes as expected when compiled unoptimized or with the **-qnoansialias** option; it successfully compiles optimized with **-qansialias**, but does not necessarily execute as expected. On non-IBM compilers, the following code may execute properly, even though it is incorrect.

```
1  extern int y = 7.;
2
3  void main() {
4      float x;
5      int i;
6      x = y;
```

```

7      i = *(int *) &x;
8      printf("i=%d. x=%f.\n", i, x);
9  }

```

In this example, the value in object `x` of type `float` has its stored value accessed via the expression `*(int *) &x`. The access to the stored value is done by the `*` operator, operating on the expression `(int *) &x`. The type of that expression is `(int *)`, which is not covered by the list of valid ways to access the value in the ISO standard, so the program violates the standard.

When **-qansialias** (the default) is in effect, the compiler front end passes aliasing information to the optimizer that, in this case, an object of type `float` could not possibly be pointed to by an `(int *)` pointer (that is, that they could not be aliases for the same storage). The optimizer performs optimization accordingly. When it compares the instruction that stores into `x` and the instruction that loads out of `*(int *)`, it believes it is safe to put them in either order. Doing the load before the store will make the program run faster, so it interchanges them. The program becomes equivalent to:

```

1  extern int y = 7.;
2
3  void main() {
4      float x;
5      int i;
6      int temp;
7      temp = *(int *) &x; /* uninitialized */
8      x = y;
9      i = temp;
10     printf("i=%d. x=%f.\n", i, x);
11 }

```

The value stored into variable `i` is the old value of `x`, before it was initialized, instead of the new value that was intended. IBM compilers apply some optimizations more aggressively than some other compilers so correctness is more important.

Related information

- [“-qlanglvl \(-std\)” on page 75](#)
- [“-O, -qoptimize” on page 44](#)

-qarch

Category

[Optimization and tuning](#)

Pragma equivalent

None.

Purpose

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

Syntax

➤ -q — arch — 

Defaults

-qarch=10

Parameters

n

Specifies the group to which a model number belongs.

The following groups of models are supported:

5

Produces code that uses instructions available on the 2064-xxx (z900) and 2066-xxx (z800) models in z/Architecture® mode.

6

Produces code that uses instructions available on the 2084-xxx (z990) and 2086-xxx (z890) models in z/Architecture mode.

7

Produces code that uses instructions available on the 2094-xxx (IBM System z9® Business Class) and 2096-xxx (IBM System z9 Business Class) models in z/Architecture mode.

8

Produces code that uses instructions available on the 2097-xxx (IBM System z10® Enterprise Class) and 2098-xxx (IBM System z10 Business Class) models in z/Architecture mode.

9

Produces code that uses instructions available on the 2817-xxx (IBM zEnterprise® 196 (z196)) and 2818-xxx (IBM zEnterprise 114 (z114)) models in z/Architecture mode.

10

Is the default value. Produces code that uses instructions available on the 2827-xxx (IBM zEnterprise EC12 (zEC12)) and 2828-xxx (IBM zEnterprise BC12 (zBC12)) models in z/Architecture mode.

11

Produces code that uses instructions available on the 2964-xxx (IBM z13® (z13)) and the 2965-xxx (IBM z13s® (z13s)) models in z/Architecture mode.

12

Produces code that uses instructions available on the 3906-xxx (IBM z14) models in z/Architecture mode.

13

Produces code that uses instructions available on the 8561-xxx (IBM z15) models in z/Architecture mode.

Usage

For any given **-qarch** setting, the compiler defaults to a specific, matching **-qtune** setting, which can provide additional performance improvements. For detailed information about using **-qarch** and **-qtune** together, see [“-qtune” on page 96](#).

Predefined macros

None.

Related information

- [“-qfloat” on page 69](#)
- [“-qtune” on page 96](#)

-qascii

Category

[“Portability and migration” on page 25](#)

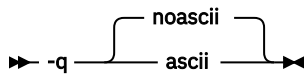
Pragma equivalent

None.

Purpose

Converts character and string literals to ISO8859-1 and enables your application to process ASCII data natively at execution time.

Syntax



Defaults

-qnoascii

Usage

When **-qascii** is in effect, the compiler performs the following:

- Uses ISO8859-1 for its default code page rather than IBM-1047 for character constants and string literals.
- Sets a flag in the program control block to indicate that the compile unit is ASCII.

When **-qnoascii** is in effect, the compiler uses the IBM-1047 code page for character constants and string literals, unless the code page is affected by other related options.

Use the **-qascii** option and the ASCII version of the runtime library if your application must process ASCII data natively at execution time.

Note: All string literals that are related to the `std::error_category` and `std::exception` classes are in EBCDIC.

Predefined macros

When **-qascii** is in effect, `__CHARSET_LIB` is defined to 1 and `_ENHANCED_ASCII_EXT` is defined to 0x410A0000. When **-qnoascii** is in effect, those macros are not defined.

-qasm (-fasm)

Category

[Language element control](#)

Pragma equivalent

None.

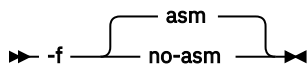
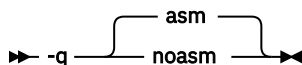
Purpose

Controls the interpretation and subsequent generation of code for assembler language extensions.

When **-fasm** (**-qasm**) is in effect, the compiler generates code for assembly statements in the source code. Suboptions specify the syntax used to interpret the content of the assembly statement.

Note: The system assembler program must be available for this command to take effect.

Syntax



Defaults

-qasm or **-fasm**

Usage

C At language levels **stdc89** and **stdc99**, token **asm** is not a keyword. At all the other language levels, token **asm** is treated as a keyword. **C**

C++ The tokens **asm**, **__asm**, and **__asm__** are keywords at all language levels. **C++**

Example

The following code snippet shows an example of the GCC conventions for **asm** syntax in inline statements:

```
int a, b;
int main() {
    asm(" AR %0,%1 " : "+r"(a) : "r"(b));
}
```

Related information

- [“-qlanglvl \(-std\)” on page 75](#)
- [“-qasmlib” on page 57](#)

-qasmlib

Category

[Input control](#)

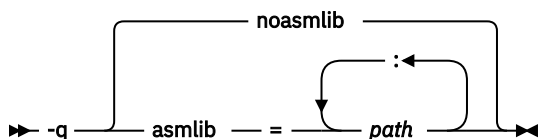
Pragma equivalent

None.

Purpose

Specifies assembler macro libraries to be used when assembling the assembler source code.

Syntax



Default

-qnoasmlib

Parameters

path

The specified macro library can be a z/OS UNIX System Services file system directory. If the suboption is a z/OS UNIX System Services file system directory.

Usage

Libraries specified with the **-qasm1ib** option or `asm1ib` XL C/C++ V2.4.1 configuration file attribute are dynamically allocated in the order in which they were specified.

Multiple specifications of **-qasm1ib** result in macro libraries being appended to the list of macro libraries in the order in which they were specified.

Specify `sys1.maclib` with **-qasm1ib** if system macros are used.

-qnoasm1ib clears the macro library concatenation.

Related information

For more information about enabling assembler code processing, see [“-qasm \(-fasm\)”](#) on page 56.

-qassert

Category

[Optimization and tuning](#)


Pragma equivalent

None.

Purpose

Enables optimizations for restrict qualified pointers.

Syntax

►► -q — assert — = — 

Defaults

-qassert=restrict

Parameters

restrict

Optimizations based on restrict qualified pointers are enabled.

norestrict

Optimizations based on restrict qualified pointers are disabled.

Usage

Restrict qualified pointers were introduced in the C99 Standard and provide exclusive initial access to the object that they point to. This means that two restrict qualified pointers, declared in the same scope, designate distinct objects and thus should not alias each other (in other words, they are disjoint). The compiler can use this aliasing in optimizations that may lead to additional performance gains.

Optimizations based on restrict qualified pointers will occur unless the user explicitly disables them with the option **-qassert=norestrict**.

-qassert=restrict does not control whether the keyword restrict is a valid qualifier or not.

You are responsible for ensuring that if a restrict pointer *p* references an object *A*, then within the scope of *p*, only expressions based on the value of *p* are used to access *A*. A violation of this rule is not diagnosed by the compiler and may result in incorrect results. This rule only applies to **-qassert=restrict**.

Predefined macros

None.

Related information

[“-qlanglvl \(-std\)” on page 75](#)

-qchars (-fsigned-char, -funsigned-char)

Category

[Floating-point and integer control](#)

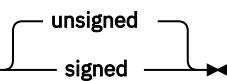
Pragma equivalent

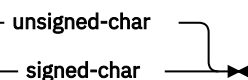
None.

Purpose

Determines whether all variables of type char is treated as signed or unsigned.

Syntax

►► -q — chars — = — 

►► -f — 

Defaults

-qchars=unsigned or **-funsigned-char**

Parameters

unsigned

Variables of type char are treated as unsigned char.

signed

Variables of type char are treated as signed char.

Usage

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.

Predefined macros

- `__CHAR_SIGNED` and `__CHAR_SIGNED__` are defined to 1 when **-qchars=signed** (**-fsigned-char**) is in effect; otherwise, they are undefined.

- `_CHAR_UNSIGNED` and `__CHAR_UNSIGNED__` are defined to 1 when **-qchars=unsigned** (**-funsigned-char**) is in effect; otherwise, they are undefined.

Example

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

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

-qcompact

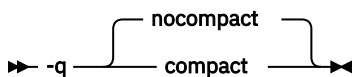
Category

[Optimization and tuning](#)

Purpose

Avoids optimizations that increase code size.

Syntax



Defaults

-qnocompact

Usage

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

This option takes effect only when it is specified at the **-O2** optimization level, or higher.

Predefined macros

`__OPTIMIZE_SIZE__` is predefined to 1 when **-qcompact** and an optimization level are in effect. Otherwise, it is undefined.

Example

To compile `myprogram.c`, instructing the compiler to reduce code size whenever possible, enter the following command:

```
xlclang myprogram.c -O -qcompact
```

-qcompress

Category

[Object code control](#)

Pragma equivalent

None.

Parameters

qualifier

Enables the compiler to generate long CSECT names.

Usage

When the CSECT option is in effect, the compiler should ensure that the code and static data sections of your object module are named. Use this option if you will be using SMP/E to service your product and to aid in debugging your program.

Specifying **-qcsect=** is equivalent to specify **-qcsect**.

The CSECT option names sections of your object module differently depending on whether you specified **-qcsect** with or without a qualifier.

The **-qcsect** option names the code and static data sections of your object module as the following:

basename#suffix

If you specify the **-qcsect** option without the **qualifier** suboption.

qualifier#basename#suffix

If you specify the **-qcsect** option with the **qualifier** suboption.

where:

qualifier

Is the suboption you specified as a qualifier

basename

Is one of the following:

- When **-c** is specified, it is the source file name with path information and the right-most extension information removed.
- When **-o** is specified, it is the output file name with path information and the right-most extension information removed.

suffix

suffix is:

- C for code CSECT.
- S for static CSECT.

Notes:

1. The # that is appended as part of the #C or #S suffix is not locale-sensitive.
2. The string that is specified as the **qualifier** suboption has the following restrictions:
 - Leading and trailing blanks are removed
 - You can specify a string of any length. However if the complete CSECT name exceeds 1024 bytes, it is truncated starting from the left.

The CSECT names for all the sections (including the code and static data sections) must conform to the following rules:

- The first character must be an alphabetic character. An alphabetic character is a letter from A through Z, or from a through z, or _, \$(code point X'5B'), #(code point X'7B') or @(code point X'7C'). The other characters in the CSECT name may be alphabetic characters, digits, or a combination of the two.
- No other special characters may be included in the CSECT name.
- No spaces are allowed in the CSECT name.
- No double-byte data is allowed in the CSECT name.

Predefined macros

None.

-qdebug

Category

Error checking and debugging

Pragma equivalent

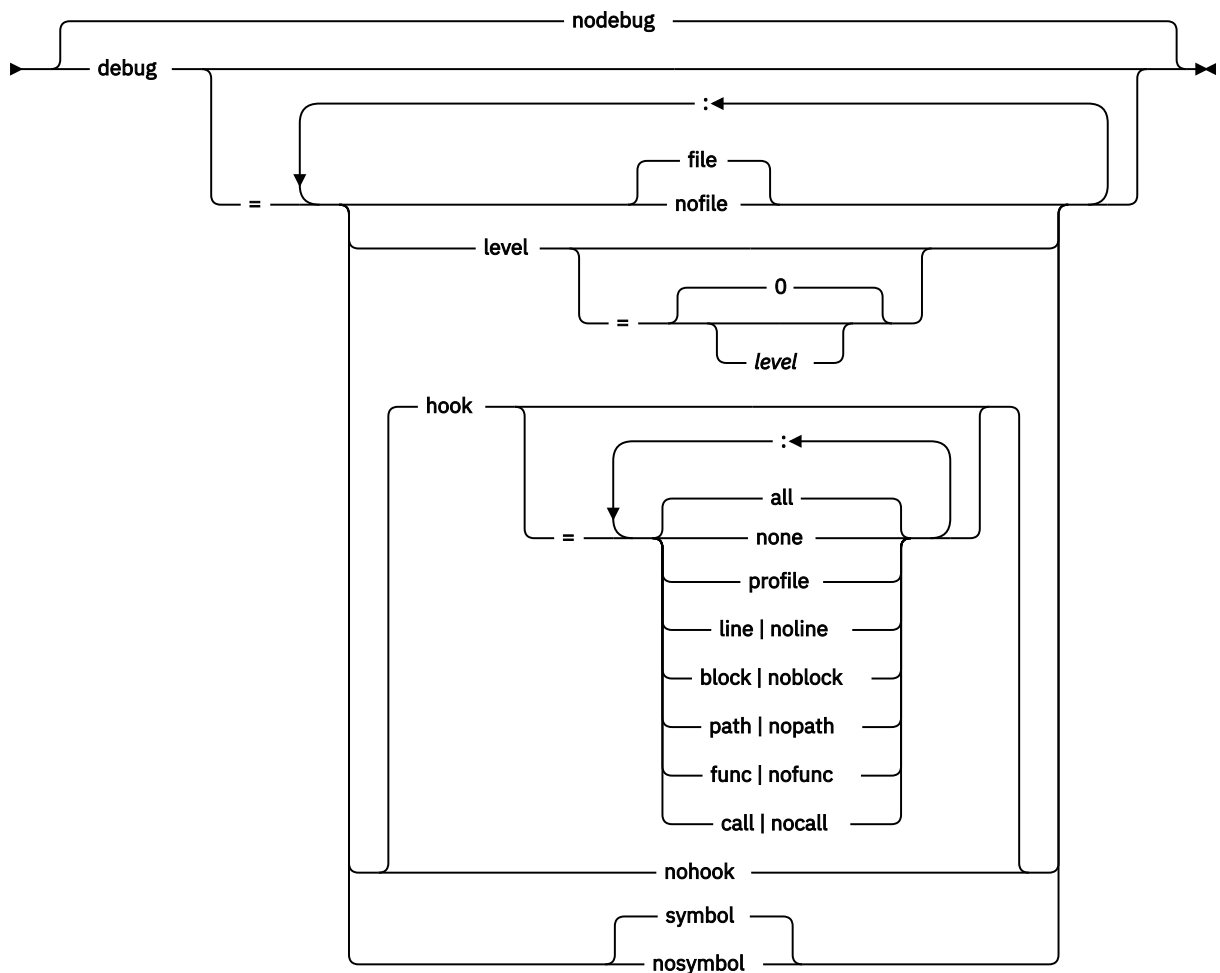
None.

Purpose

Instructs the compiler to generate debugging information.

Syntax

►► -q ►



Defaults

- **-qnodebug**
- For **file**, the default is **file**.
- For **level**, the default is **level=0**.
- For **hook**, the defaults are **hook=all** for **-qnooptimize** and **hook=none:profile** for **-qoptimize**.
- For **symbol**, the default is **symbol**.

Parameters

file | **nofile**

Controls whether the DWARF debugging information is stored in a separate debug file.

The **file** suboption specifies the name of the output file for **-qdebug**. The output file can be a z/OS UNIX file or a z/OS UNIX System Services directory.

The **nofile** suboption instructs the compiler to place the debugging information in the GOFF NOLOAD classes in the object file instead of a separate debug side file. The binder then merges the debugging information from different object files into the NOLOAD classes in the executable or library at binding time. The debugging information in the NOLOAD classes will be loaded only when it is explicitly required by the debugger.

If you do not specify a file name with the **file** suboption, the compiler stores the debugging information in a file that has the name of the source file with a .dbg extension.

For a z/OS UNIX file system directory compile, the **file** option specifies the z/OS UNIX file system directory where the output files are generated.

The compiler resolves the full path name for this file name, and places it in the generated object file. This information can be used by program analysis tools to locate the output file for **-qdebug**. You can examine this generated file name in the compiler listing file (see “-qlist” on page 78 for instructions on how to create a compiler listing file), as shown in the following example:

```
PPA4: Compile Unit Debug Block
000140 0000001A      =F'26'      DWARF File Name
000144 ****      C'/hfs/fullpath/filename.dbg'
```

If the compiler cannot resolve the full path name for the file name (for example, because the search permission was denied for a component of the file name), the compiler will issue a warning message, and the relative file name will be used instead.

level

Controls the amount of debugging information produced. Different levels can balance between debug capability and compiler optimization. Higher levels provide more complete debug support, at the cost of runtime or possible compile-time performance. Lower levels provide higher runtime performance, at the cost of some capability in the debugging session. The **level** suboption has the following values:

0

- If the **-qoptimize** compiler option is specified, **-qdebug=level=0** is equivalent to **-qdebug=level=2**.
- If the **-qnooptimize** compiler option is specified, **-qdebug=level=0** is equivalent to **-qnodebug**.

Note: In the z/OS UNIX System Services environment, **-g** forces **-qnooptimize** and translates to **-qdebug=level=0**. To debug at an optimization level, you must specify **-g** with an explicit level.

1

Generates minimal read-only debugging information about line numbers and source file names. No program state is preserved.

Note: Specifying **-qdebug=level=1** is equivalent to specifying **-qnodebug** with **-qgonumber**. If **-qdebug=level=1** and **-qnogonumber** are specified, a warning message is issued, and the options are set to **-qnodebug** and **-qnogonumber**.

2

Generates read-only debugging information about line numbers, source file names, and symbols. When **-qoptimize=2** or higher level is specified, no program state is preserved.

3, 4

Generates read-only debugging information about line numbers, source file names, and symbols.

When **-qoptimize=2** or higher level is specified:

- No program state is preserved.
- Function parameter values are available to the debugger at the beginning of each function.

Note: **-qdebug=level=3** implies STOREARGS if the linkage mode is XPLINK.

5, 6, 7

Generates read-only debugging information about line numbers, source file names, and symbols.

When **-qoptimize=2** or higher level is specified:

- Program state is available to the debugger at if constructs, loop constructs, procedure calls, and function calls.
- Function parameter values are available to the debugger at the beginning of each function.

8

Generates read-only debugging information about line numbers, source file names, and symbols.

When **-qoptimize=2** or higher level is specified:

- Program state is available to the debugger at the beginning of every executable statement.
- Function parameter values are available to the debugger at the beginning of each function.

9

Generates debugging information about line numbers, source file names, and symbols. Modifying the value of a variable in the debugger is allowed and respected.

When **-qoptimize=2** or higher level is specified:

- Program state is available to the debugger at the beginning of every executable statement.
- Function parameter values are available to the debugger at the beginning of each function.

hook

Note: If the **-qoptimize** compiler option is specified, the only valid suboptions for **-qhook** are **call** and **func**. If other suboptions are specified, they will be ignored.

Controls the generation of **line**, **block**, **path**, **call**, and **func** hook instructions. Hook instructions appear in the compiler Pseudo Assembly listing in the following form:

```
EX r0,H00K..[type of hook]
```

The type of hook that each hook suboption controls is summarized in the list below:

• **line**

- STMT - General statement

• **block**

- BLOCK-ENTRY - Beginning of block
- BLOCK-EXIT - End of block
- MULTIEXIT - End of block and procedure

• **path**

- LABEL - A label
- DOBGN - Start of a loop
- TRUEIF - True block for an if statement
- FALSEIF - False block for an if statement
- WHENBGN - Case block
- OTHERW - Default case block

- GOTO - Goto statement
- POSTCOMPOUND - End of a PATH block
- **call**
 - CALLBGN - Start of a call sequence
 - CALLRET - End of a call sequence
- **func**
 - PGM-ENTRY - Start of a function
 - PGM-EXIT - End of a function

There is also a set of shortcuts for specifying a group of hooks:

none

It is the same as specifying **noline**, **noblock**, **nopath**, **nocall**, and **nofunc**. It instructs the compiler to suppress all hook instructions.

all

It is the same as specifying **line**, **block**, **path**, **call**, and **func**. It instructs the compiler to generate all hook instructions. This is the ideal setting for debugging purposes.

profile

It is the same as specifying **call** and **func**. It is the ideal setting for tracing the program with the Performance Analyzer.

symbol

This option provides you with access to variable and other symbol information. For optimized code, the results are not always well-defined for every variable because the compiler might have optimized away their use.

Usage

When the **-qdebug** option is in effect, the compiler generates debugging information based on the DWARF Version 4 debugging information format, which has been developed by the UNIX International Programming Languages Special Interest Group (SIG), and is an industry standard format. The generated debugging information is stored in the file specified by the **file** suboption, or in GOFF NOLOAD classes when the **nofile** suboption is specified.

If you specify the **-qinline** and **-qdebug** compiler options when **-qoptimize** is in effect, the inline debugging information is generated for inline procedures as well as parameters and local variables of inline procedures.

If you specify the **-qinline** and **-qdebug** compiler options when **-qnooptimize** is in effect, **-qinline** is ignored.

When **-qoptimize=2** or **-qoptimize=3** is used with **-qdebug**, the **-qdebug=symbol** suboption is enabled by default.

In the z/OS UNIX System Services environment, **-g** forces **-qdebug**, **-qnooptimize**, and **-qgonumber**.

If you specify **-qdebug**, automonitor debugging information is generated to list the variables that occur on each statement of the program source file.

Predefined macros

None.

Example

If you specify **-qdebug** and **-qnodebug** multiple times, the compiler uses the last specified option with the last specified suboption. For example, the following specifications have the same result:

```
xlclang -qdebug=level=8 -qnodebug -qdebug=nosymbol hello.c
```

```
xlclang -qdebug=level=8:nosymbol hello.c
```

-qdigraph

Category

[Language element control](#)

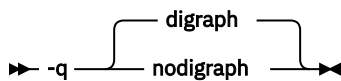
Pragma equivalent

None.

Purpose

Enables recognition of digraph key combinations **C++** and operator keywords **C++** to represent characters that are not found on some keyboards. Digraph key combinations include <:, <%, and so on. **C++** Operator keywords include and, or, and so on. **C++**

Syntax



Defaults

-qdigraph

Usage

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

Predefined macros

None.

Examples

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

```
xlclang myprogram.c -qnodigraph
```

Related information

- [“-qlanglvl \(-std\)” on page 75](#)

-qeh (C++ only)

Category

[Object code control](#)

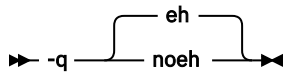
Pragma equivalent

None.

Purpose

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

Syntax



Defaults

-qeh

Usage

When **-qeh** is in effect, you can control the generation of C++ exception handling code.

When the **-qnoeh** option is in effect, the generation of the exception handling code is suppressed, which results in code that runs faster, but it will not be ANSI-compliant if the program uses exception handling.

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

If you compile a source file with **-qnoeh**, active objects on the stack are not destroyed if the stack collapses in an abnormal fashion. For example, if a C++ object is thrown, or a Language Environment exception or signal is raised, objects on the stack will not have their destructors run. If **-qnoeh** has been specified and the source file has try/catch blocks or throws objects, the program may not execute as expected.

Predefined macros

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

-qexportall

Category

[Object code control](#)

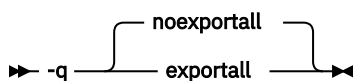
Pragma equivalent

None.

Purpose

Exports all externally defined functions and variables in the compilation unit so that a DLL application can use them.

Syntax



calculations performed at compile time or on other types of computers. Negative zero results may be produced. This option may affect the precision of floating-point intermediate results.

Note: The suboption **maf** does not have any effect on extended floating-point operations.

rrm | norrm

Runtime Rounding Mode (RRM) prevents floating-point optimizations that are incompatible with runtime rounding to plus and minus infinity modes. It informs the compiler that the floating-point rounding mode may change at run time or that the floating-point rounding mode is not round to nearest at run time.

afp | noafp

afp instructs the compiler to generate code which uses the full complement of 16 floating point registers. These include the four original floating-point registers, numbered 0, 2, 4, and 6, and the Additional Floating Point (AFP) registers, numbered 1, 3, 5, 7 and 8 through 15.

noafp limits the compiler to generating code using only the original four floating-point registers, 0, 2, 4, and 6, which are available on all *IBM Z*® machine models.

Usage

This option allows controlling floating point accuracy and use of floating point registers.

Using **-qfloat** suboptions other than the default settings might produce incorrect results in floating-point computations if the system does not meet all required conditions for a given suboption. Therefore, use this option only if the floating-point calculations involving IEEE floating-point values are manipulated and can properly assess the possibility of introducing errors in the program.

If **-qstrict** or **-qnostrict** and **float** suboptions conflict, the last setting specified is used.

Predefined macros

`__BFP__` is defined to 1 when **-qfloat** is in effect.

Examples

To compile `myprogram.c` so that the constant floating-point expressions are evaluated at compile time and multiply-add instructions are not generated, enter:

```
xlclang myprogram.c -qfloat=fold:nomaf
```

Related information

- [“-qarch” on page 54](#)
- [“-qstrict” on page 92](#)

-qgonumber

Category

[Error checking and debugging](#)

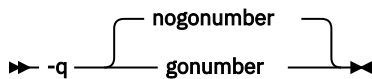
Pragma equivalent

None.

Purpose

Generates line number tables that correspond to the input source file for Debug Tool and CEEDUMP processing.

Syntax



Defaults

- **-qnogonumber**
- If **-qdebug** is specified, **-qgonumber** is the default.

Usage

The **-qgonumber** option is active by default when you use the **-qdebug** option. The **-qdebug** option will activate the **-qgonumber** option unless **-qnogonumber** has been explicitly specified.

In the z/OS UNIX System Services environment, the **-qgonumber** option is enforced when you use the **-g** flag option using the **xlclang** or **xlclang++** command. In another words, the **-g** flag option will always activate the **-qgonumber** option, regardless of other option specifications.

Predefined macros

None.

Related information

[“-qdebug” on page 63](#)

-qhalt

Category

[Error checking and debugging](#)

Purpose

Stops compilation process of a set of source code before producing any object, executable, or assembler source files if the maximum severity of compile-time messages equals or exceeds the severity specified for this option.

Syntax

```
➡ -q — halt — = — num ➡
```

Defaults

-qhalt=16

Parameters

num

Return code from the compiler.

Usage

If the return code from compiling a particular member is greater than or equal to the value *num* specified in the **-qhalt** option, no more members are compiled. This option only applies to the compilation of all members of a specified z/OS UNIX System Services file system directory.

Predefined macros

None.

-qignerrno

Category

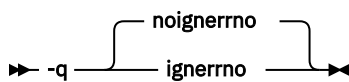
[Optimization and tuning](#)

Purpose

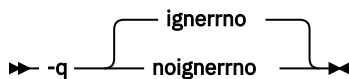
Allows the compiler to perform optimizations as if system calls would not modify `errno`.

Syntax

For **-qnooptimize** and **-qoptimize=2**:



For **-qoptimize=3**:



Defaults

- **-qnoignerrno** when **-qnooptimize** or **-qoptimize=2** is in effect.
- **-qignerrno** when **-qoptimize=3** is in effect.

Usage

When the **-qignerrno** compiler option is in effect, the compiler is informed that your application is not using `errno`. Specifying this option allows the compiler to explore additional optimization opportunities for library functions in LIBANSI. The input to the library functions is assumed to be valid. Invalid input can lead to undefined behavior.

When the **-qnoignerrno** compiler option is in effect, the compiler assumes that your application is using `errno`.

ANSI library functions use `errno` to return the error condition. If your program does not use `errno`, the compiler has more freedom to explore optimization opportunities for some of these functions (for example, `sqrt()`). You can control this optimization by using the **-qignerrno** option.

The **-qignerrno** option is turned on by **-qoptimize=3**. You can use **-qnoignerrno** to turn it off if necessary.

Predefined macros

`__IGNERRNO__` is defined to 1 when **-qignerrno** is in effect; otherwise, it is undefined.

Related information

- [“-O, -qoptimize” on page 44](#)

-qinclude

Category

Input control

Pragma equivalent

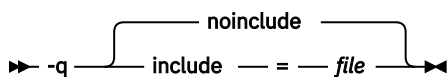
None.

Purpose

Specifies additional header files to be included in a compilation unit, as though the files were named in an `#include` statement in the source file.

The headers are inserted before all code statements and any headers specified by an `#include` preprocessor directive in the source file. This option is provided for portability among supported platforms.

Syntax



Defaults

None.

Parameters

file

The header file to be included in the compilation units being compiled.

Usage

Firstly, *file* is searched in the preprocessor's working directory. If *file* is not found in the preprocessor's working directory, it is searched for in the search chain of the **#include** directive. If multiple **-qinclude** options are specified, the files are included in order of appearance on the command line.

Predefined macros

None.

Examples

To include the files `test1.h` and `test2.h` in the source file `test.c`, enter the following command:

```
xlclang -qinclude test1.h -qinclude test2.h test.c
```

Related information

- [“Directory search sequence for included files” on page 6](#)

-qinline

Category

Optimization and tuning

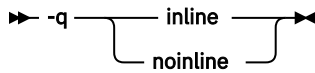
Pragma equivalent

None.

Purpose

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

Syntax



Defaults

If **-qnooptimize** is specified, the default is **-qnoinline**; otherwise, the default is **-qinline**.

Usage

You can specify **-qinline** with any optimization level of **-O**, **-O2**, **-O3** to enable inlining of functions, including those functions that are declared with the `inline` specifier **C++** or that are defined within a class declaration **C++**.

When **-qinline** is in effect, the compiler determines whether inlining a specific function can improve performance. That is, whether a function is appropriate for inlining is subject to two factors: limits on the number of inlined calls and the amount of code size increase as a result. Therefore, enabling inlining a function does not guarantee that function will be inlined.

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

Specifying **-qnoinline** disables all inlining, including functions declared explicitly as `inline`. However, the **-qnoinline** option does not affect the inlining of the following functions:

- **IBM** Functions that are specified with the `always_inline` or `__always_inline__` attribute **IBM**

If you specify the **-g** option to generate debugging information, the inlining effect of **-qinline** might be suppressed.

If you specify the **-qcompact** option to avoid optimizations that increase code size, the inlining effect of **-qinline** might be suppressed.

Predefined macros

None.

Examples

To compile `myprogram.c` so that no functions are inlined, use the following command:

```
xlclang myprogram.c -O2 -qnoinline
```

Related information

- [“-g” on page 34](#)
- [“-O, -qoptimize” on page 44](#)
- [“Compiler listings” on page 10](#)

-qlanglvl (-std)

Category

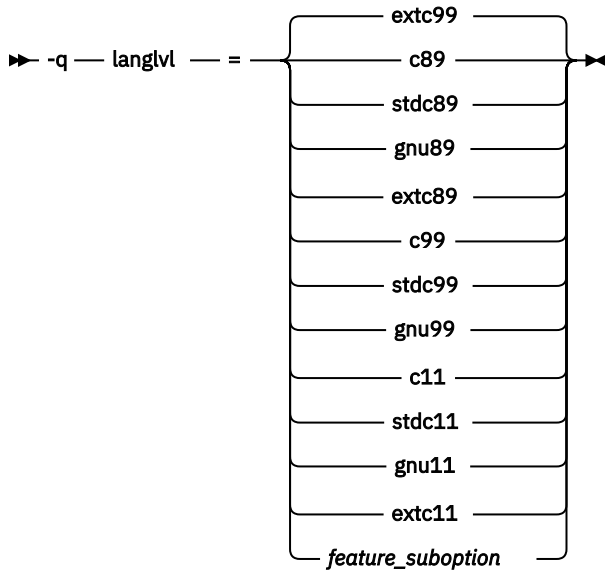
Language element control

Purpose

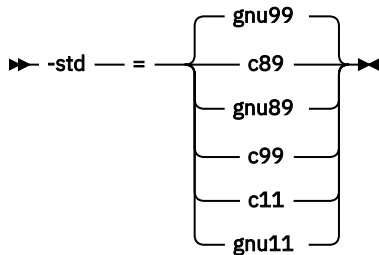
Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.

Syntax

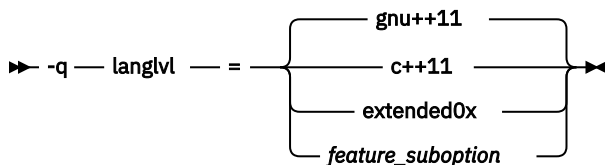
-qlanglvl syntax (C only)



-std syntax (C only)



-qlanglvl syntax (C++ only)



-std syntax (C++ only)



Defaults

- **C** -qlanglvl=extc99 or -std=gnu99
- **C++** -qlanglvl=gnu++11 or -std=gnu++11

Parameters

C Parameters for C language programs:

c89 | **stdc89**

Use it if you are compiling new or ported code that is ISO C compliant. It indicates language constructs that are defined by ISO.

gnu89 | **extc89**

Indicates language constructs that are defined by the ISO C89 standard, plus additional orthogonal language extensions that do not alter the behavior of this standard.

Note: The unicode literals are enabled under the EXTC89 language level, and disabled under the strictly-conforming language levels. When the unicode literals are enabled, the macro `__IBM_UTF_LITERAL` is predefined to 1. Otherwise, this macro is not predefined.

c99 | **stdc99**

Compilation conforms strictly to the ISO C99 standard, also known as ISO C99.

gnu99 | **extc99**

Indicates language constructs that are defined by the ISO C99 standard, plus additional orthogonal language extensions that do not alter the behavior of the standard.

Note: The unicode literals are enabled under the extc99 language level, and disabled under the strictly-conforming language levels. When the unicode literals are enabled, the macro `__IBM_UTF_LITERAL` is predefined to 1. Otherwise, this macro is not predefined.

C11 **c11** | **stdc11**

Compilation conforms strictly to the ISO C11 standard. **C11**

C11 **gnu11** | **extc11**

Compilation is based on the C11 standard, invoking all the currently supported C11 features and other implementation-specific language extensions. **C11**

The following table reflects the mapping between the **-qlanglvl** and **-std** suboptions:

Table 18. Mapping between the -qlanglvl and -std suboptions (C only)	
-qlanglvl suboption	Mapping to -std suboption
c89 stdc89	c89
gnu89 extc89	gnu89
c99 stdc99	c99
gnu99 extc99	gnu99
c11 stdc11	c11
gnu11 extc11	gnu11

C

C++ Parameters for C++ language programs:

C++11 **c++11**

Compilation conforms strictly to the ISO C++ standard plus amendments, also known as ISO C++11.

C++11

C++11 **gnu++11 | extended0x**

Compilation is based on the ISO C++ standard, with some differences to accommodate extended language features. **C++11**

The following table reflects the mapping between the **-qlanglvl** and **-std** suboptions:

Table 19. Mapping between the -qlanglvl and -std suboptions (C ++ only)	
-qlanglvl suboption	Mapping to -std suboption
c++11	c++11
gnu++11 extended0x	gnu++11

C++

The following feature suboption is available:

libext | nolibext

Specifying this option affects the headers provided by the C runtime library, which in turn control the availability of general ISO runtime extensions. In addition, it also defines the following macros and sets their values to 1:

- **_MI_BUILTIN** (this macro controls the availability of machine built-in instructions.
- **_EXT** (this macro controls the availability of general ISO runtime extensions)

The default is **-qlanglvl=libext**. However, **-qlanglvl=libext** is implicitly enabled by:

```
-qlanglvl=extc89:extc99
```

Predefined macros

See “[Macros related to language levels](#)” on page 118 for a list of macros that are predefined by **-qlanglvl** suboptions.

-qlibansi

Category

[Optimization and tuning](#)

Pragma equivalent

None.

Purpose

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

When **-qlibansi** is in effect, the optimizer can generate better code because it will know about the behavior of a given function, such as whether or not it has any side effects.

Syntax

➤ **-q** nolibansilibansi ➤

Defaults

-qnolibansi

Predefined macros

`__LIBANSI__` is defined to 1 when **-qlibansi** is in effect; otherwise, it is not defined.

-qlist

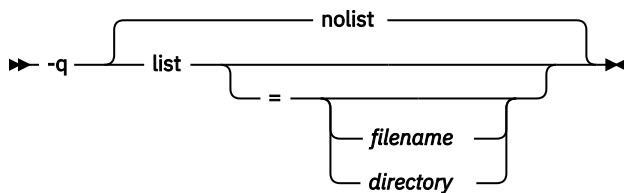
Category

Listings, messages, and compiler information

Purpose

Produces a compiler listing file that includes object and constant area sections.

Syntax



Defaults

-qnolist

Parameters

filename

Specifies the z/OS UNIX System Services file name for the compiler listing.

directory

Specifies the z/OS UNIX System Services directory for the compiler listing.

Usage

When **-qlist** is in effect, the listing directs to stdout.

When **-qlist=** is in effect, a listing file is generated with a `.lst` suffix for each source file named on the command line.

When **-qlist=filename** is in effect, a listing file is generated to `filename`.

When **-qlist=directory** is in effect, a listing file with a `.lst` suffix is generated in the directory.

You can use the object or assembly listing to help understand the performance characteristics of the generated code and to diagnose execution problems.

Predefined macros

None.

Example

To compile `myprogram.c` and to produce a listing (`.lst`) file that includes object , enter:

```
xlclang myprogram.c -qlist=
```

Related information

[“Compiler listings” on page 10](#)

-qmakedep

Category

Output control

Pragma equivalent

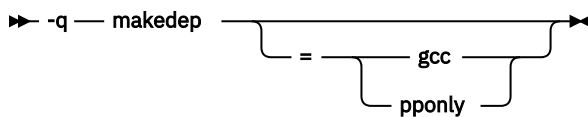
None.

Purpose

Produces the dependency files that are used by the make tool for each source file.

The dependency output file is named with a `.d` suffix.

Syntax



Defaults

Not applicable.

Parameters

gcc

The format of the generated **make** rule to match the GCC format: the dependency output file includes a single target that lists all of the main source file's dependencies.

This suboption is equivalent to **-MD**.

pponly

Instructs the compiler to produce only the make dependency files without generating an object file, with the same make dependency file format as the format produced by the **gcc** suboption.

Usage

For each source file with a `.c`, `.C`, `.cpp`, or `.i` suffix that is named on the command line, a dependency output file is generated with the same name as the object file but with a `.d` suffix. Dependency output files are not created for any other types of input files. If you use the **-o** option to rename the object file, the name of the dependency output file is based on the name specified in the **-o** option.

The dependency output files generated by these options are not **make** description files; they must be linked before they can be used with the **make** command. For more information about this command, see your operating system documentation.

The output file look like:

```
t.o: t.c /usr/include/c++/stdio.h \
      /usr/include/c++/__config \
      /usr/include/le/features.h \
      /usr/include/le/stdio.h \
      /usr/include/le/sys/types.h
```

Include files are listed according to the search order rules for the `#include` preprocessor directive, described in [“Directory search sequence for included files”](#) on page 6. If the include file is not found, it is not added to the `.d` file.

Files with no include statements produce dependency output files that contain one line listing only the input file name.

Predefined macros

None.

Examples

Example 1: To compile `mysource.c` and create a dependency output file named `mysource.d`, enter:

```
xlclang -c -qmakedep mysource.c
```

Example 2: To compile `my_src.c` and create a dependency output file named `mysource.d`, enter:

```
xlclang -c -qmakedep my_src.c -MF mysource.d
```

Example 3: To compile `my_src.c` and create a dependency output file named `mysource.d` in the `deps/` directory, enter:

```
xlclang -c -qmakedep my_src.c -MF deps/mysource.d
```

Example 4: To compile `my_src.c` and create an object file named `my_obj.o` and a dependency output file named `my_obj.d`, enter:

```
xlclang -c -qmakedep my_src.c -o my_obj.o
```

Example 5: To compile `my_src.c` and create an object file named `my_obj.o` and a dependency output file named `mysource.d`, enter:

```
xlclang -c -qmakedep my_src.c -o my_obj.o -MF mysource.d
```

Example 6: To compile `my_src1.c` and `my_src2.c` to create two dependency output files, named `my_src1.d` and `my_src2.d` respectively, enter:

```
xlclang -c -qmakedep my_src1.c my_src2.c
```

Related information

- “-o” on page 46
- “Directory search sequence for included files” on page 6
- The **-M**, **-MD**, **-MF**, **-MG**, **-MM**, **-MMD**, **-MP**, **-MQ**, and **-MT** options that GCC provides. For details, see [GNU Compiler Collection online documentation](http://gcc.gnu.org/onlinedocs/)(<http://gcc.gnu.org/onlinedocs/>).

-qmaxmem

Category

[Optimization and tuning](#)

Purpose

Limits the amount of memory that the compiler allocates while performing specific, memory-intensive optimizations to the specified number of kilobytes.

Syntax

► **-q** *maxmem* *size_limit* ◄

Defaults

-qmaxmem=2097152

Parameters

size_limit

The valid range for *size* is 0 to 2097152. Use the *size-limit* suboption if you want to specify a memory size of less value than the default.

Usage

If the memory specified by the **-qmaxmem** option is insufficient for a particular optimization, the compilation is completed in such a way that the quality of the optimization is reduced, and a warning message is issued.

When a large *size* is specified for **-qmaxmem**, compilation might be aborted because of insufficient virtual storage, depending on the source file being compiled, the size of the subprogram in the source, and the virtual storage available for the compilation.

The advantage of using the **-qmaxmem** option is that, for large and complex applications, the compiler produces a slightly less-optimized object module and generates a warning message, instead of terminating the compilation with an error message of “insufficient virtual storage”.

Notes:

1. The limit that is set by **-qmaxmem** is the amount of memory for specific optimizations, and not for the compiler as a whole. Tables that are required during the entire compilation process are not affected by or included in this limit.
2. Setting a large limit has no negative effect on the compilation of source files when the compiler needs less memory.
3. 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.
4. Increasing the limit does not necessarily mean that the resulting program will be faster, only that the compiler may be able to find opportunities to increase performance.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the memory specified for local table is 16384 kilobytes, enter:

```
xlclang myprogram.c -qmaxmem=16384
```

-qmemory

Category

[Compiler customization](#)



Pragma equivalent

None.

Purpose

Improves compile-time performance by using a memory file in place of a temporary work file, if possible.

Syntax

►► -q  

Defaults

-qmemory

Usage

This option generally increases compilation speed, but you may require additional memory to use it. If you use this option and the compilation fails because of a storage error, you must increase your storage size or recompile your program using the **-qnomemory** option.

Predefined macros

None.

-qoffset

Category

Listings, messages, and compiler information

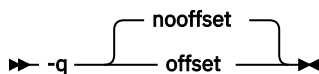
Pragma equivalent

None.

Purpose

Lists offset addresses relative to entry points of functions.

Syntax

➡ -q 

Defaults

-qnooffset

Usage

When the **-qoffset** compiler option is in effect, the compiler displays the offset addresses relative to the entry point or start of each function in the pseudo assembly listing generated by the **-qlist** option. The **-qoffset** compiler option also prints the CSECT Offset field in the pseudo assembly listing for a function, which shows the offset of the function in the CSECT.

When the **-qnooffset** compiler option is in effect, the compiler displays the offset addresses relative to the beginning of the generated code in the pseudo assembly listing generated by the **-qlist** option and does not display the entry point.

If you use the **-qoffset** option, you must also specify the **-qlist** option to generate the pseudo assembly listing. If you specify the **-qoffset** option but omit the **-qlist** option, the compiler generates a warning message, and does not produce a pseudo assembly listing.

Predefined macros

None.

Related information

[“-qlist” on page 78](#)

-qoptfile

Category

[Compiler customization](#)

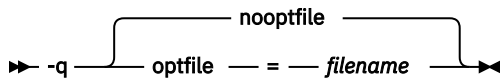
Pragma equivalent

None.

Purpose

Specifies an options file that contains a list of additional command line options to be used for the compilation.

Syntax



Defaults

-qnooptfile

Parameters

filename

Specifies the name of the options that contains a list of additional command line options. *filename* can contain a relative path or absolute path, or it can contain no path. It is a plain text file with one or more command line options per line.

Usage

The format of the options file follows these rules:

- Specify the options you want to include in the file with the same syntax as on the command line. The options file is a whitespace-separated list of options. The following special characters indicate whitespace: \n, \v, \t. (All of these characters have the same effect.)
- A character string between a pair of single or double quotation marks are passed to the compiler as one option.
- You can include comments in the options file. Comment lines start with the # character and continue to the end of the line. The compiler ignores comments and empty lines.

When processed, the compiler removes the **-qoptfile** option from the command line, and sequentially inserts the options included in the file before the other subsequent options that you specify.

Predefined macros

None.

-qphaseid, -qphsinfo

Category

[Listings, messages, and compiler information](#)

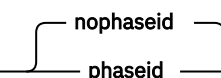
Pragma equivalent

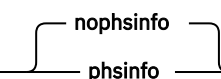
None.

Purpose

Causes each compiler component (phase) to issue an informational message as each phase begins execution and reports the time taken in each compilation phase.

Syntax

```
➤ -q  ➤
```

```
➤ -q  ➤
```

Defaults

-qnophaseid, -qnophsinfo

Usage

The informational message identifies the compiler phase module name, product identification, and build level and thus assists you with determining the maintenance level of each compiler component (phase).

Phase ID information is sent to `stdout` and the timing information is sent to `stderr`.

In the z/OS UNIX System Services environment, **-qphsinfo** is synonymous with the **-qphaseid** compiler option.

Predefined macros

None.

Example

To compile `myprogram.c` and report the time taken for each phase of the compilation, enter the following command:

```
xlclang myprogram.c -qphaseid
```

or:

```
xlclang myprogram.c -qphsinfo
```

The output looks like:

```
CLC0000(I) Product(5650-ZOS) Phase(CLCE0PTP) Level(D190213.Z231)
CLC0000(I) Product(5650-ZOS) Phase(CLCDVR ) Level(D190213.Z231)
CLC0000(I) Product(5650-ZOS) Phase(CLCECL ) Level(D190213.Z231)
=====
                        Clang front-end time report
=====
Total Execution Time: 0.0000 seconds (0.0046 wall clock)
---Wall Time---   --- Name ---
  0.0046 (100.0%) Clang front-end timer
  0.0046 (100.0%) Total
CLC0000(I) Product(5650-ZOS) Phase(CLCETBY ) Level(D190213.Z231)
CLC0000(I) Product(5650-ZOS) Phase(CLCECWI ) Level(D190213.Z231)
W-TRANS - Phase Ends; CPU sec=0.00
OPTIMIZ - Phase Ends; CPU sec=0.00
REGALLO - Phase Ends; CPU sec=0.00
AS       - Phase Ends; CPU sec=0.00
```


-qro

Category

[Object code control](#)

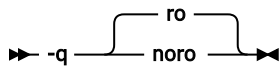
Purpose

Specifies the storage type for string literals.

When **-qro** is in effect, strings are placed in read-only memory. When **-qnoro** is in effect, strings are placed in read-write memory.

Syntax

Option syntax



Defaults

-qro

Usage

Placing string literals in read-only memory can improve runtime performance and save storage. However, code that attempts to modify a read-only string literal may generate a memory error.

Predefined macros

None.

Example

To compile `myprogram.c` so that the storage type is writable, enter:

```
xlclang myprogram.c -qnoro
```

Related information

- [“-qroconst” on page 85](#)

-qroconst

Category

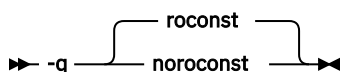
[Object code control](#)

Purpose

Specifies the storage location for constant values.

When **-qroconst** is in effect, constants are placed in read-only storage. When **-qnoroconst** is in effect, constants are placed in read/write storage.

Syntax



Defaults

-  **-qnoconst**
-  **-qroconst**

Usage

Placing constant values in read-only memory can improve runtime performance, save storage, and provide shared access. However, code that attempts to modify a read-only constant value generates a memory error.

"Constant" in the context of the **-qroconst** option refers to variables that are qualified by `const`, including `const`-qualified characters, integers, floats, enumerations, structures, unions, and arrays. The following constructs are not affected by this option:

- Variables qualified with `volatile` and aggregates (such as a structure or a union) that contain `volatile` variables
- Pointers and complex aggregates containing pointer members
- Automatic and static types with block scope
- Uninitialized types
- Regular structures with all members qualified by `const`
- Initializers that are addresses, or initializers that are cast to non-address values

The **-qroconst** option does not imply the **-qro** option. Both options must be specified if you want to specify storage characteristics of both string literals (**-qro**) and constant values (**-qroconst**).

Predefined macros

None.

Related information

- [“-qro” on page 85](#)

-qrtcheck

Category

[Error checking and debugging](#)

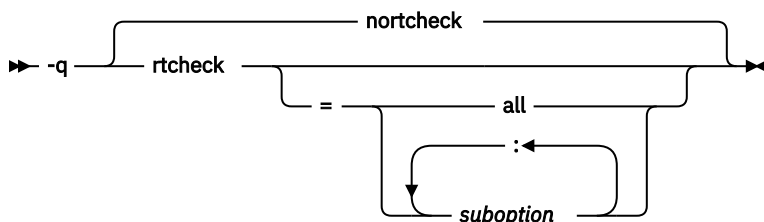
Pragma equivalent

None.

Purpose

Generates compare-and-trap instructions which perform certain types of runtime checking. The messages can help you to debug your C and C++ programs.

Syntax



Defaults

-qnocheck

Parameters

suboption is one of the suboptions that are shown in [Table 20 on page 87](#), which lists the **-qnocheck** suboptions and the messages they generate.

Note: Default suboptions are underlined.

Table 20. -qnocheck suboptions and descriptions

RTCHECK Suboption	Description
all	Automatically generates compare-and-trap instructions for all possible runtime checks. This suboption is equivalent to -qnocheck .
<u>bounds</u> nobounds	Performs runtime checking of addresses when subscripting within an object of known size.
<u>divzero</u> nodivzero	Performs runtime checking of integer division. A trap will occur if an attempt is made to divide by zero.
<u>nullptr</u> nonullptr	Performs runtime checking of addresses contained in pointer variables used to reference storage.

Usage

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

You can use the **all** suboption along with the **no...** form of one or more of the other options as a filter. For example, using:

```
xlclang -qnocheck=all:nonullptr
```

provides checking for everything except for addresses contained in pointer variables used to reference storage. If you use **all** with the **no...** form of the suboptions, **all** should be the first suboption.

Notes:

1. The **-qnocheck** option is only valid for architecture level 8 or above, and for Language Environment V1.10 and up.
2. **-qnocheck** without suboption means **-qnocheck=all**.

Predefined macros

None.

-qnocheck (-frtti) (C++ only)

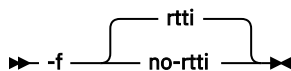
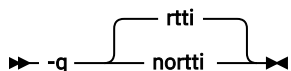
Category

[Object code control](#)

Purpose

Generates runtime type identification (RTTI) information for classes with virtual functions.

Syntax



Defaults

-qrtti (-frtti)

Usage

For improved runtime performance, suppress RTTI information generation with the **-qnortti (-fno-rtti)** setting.

You should be aware of the following effects when specifying the **-qrtti (-frtti)** compiler option:

- Contents of the virtual function table will be different when **-qrtti (-frtti)** is specified.
- When linking objects together, all corresponding source files must be compiled with the correct **-qrtti (-frtti)** option specified.
- If you compile a library with mixed objects (**-qrtti (-frtti)** specified for some objects, **-qnortti (-fno-rtti)** specified for others), you might get an undefined symbol error.

Predefined macros

- `__RTTI_ALL__` is defined to 1 when **-qrtti (-frtti)** is in effect; otherwise, it is undefined.
- `__NO_RTTI__` is defined to 1 when **-qnortti (-fno-rtti)** is in effect; otherwise, it is undefined.

Related information

[“-qlanglvl \(-std\)”](#) on page 75.

-qservice

Category

[Error checking and debugging](#)

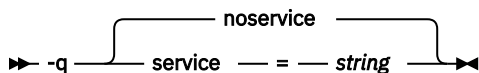
Pragma equivalent

None.

Purpose

Places a *string* in the object module, which is displayed in the traceback if the application fails abnormally.

Syntax



Defaults

-qnoservice

Parameters

string

User-specified string of characters.

Usage

When the **-qservice** compiler option is in effect, the *string* in the object module is loaded into memory when the program is executing. If the application fails abnormally, the *string* is displayed in the traceback.

You must enclose your *string* in quotation marks.

The following restrictions apply to the *string* specified:

- The *string* cannot exceed 64 characters in length. If it does, excess characters are removed, and the *string* is truncated to 64 characters.
- All characters, including DBCS characters, are valid as part of the *string*.
- Only characters which belong to the invariant character set should be used, to ensure that the signature within the object module remains readable across locales.

Predefined macros

None.

-qshowmacros

Category

[“Output control” on page 17](#)

Pragma equivalent

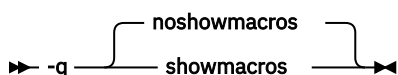
None.

Purpose

Emits macro definitions to preprocessed output.

Emitting macros to preprocessed output can help determine functionality available in the compiler. The macro listing may prove useful for debugging complex macro expansions, as well.

Syntax



Defaults

-qnoshowmacros

Usage

Note the following when using this option:

- This option has no effect unless preprocessed output is generated; for example, by using the **-E** or **-P** options.
- If a macro is defined and subsequently undefined before compilation ends, this macro will not be included in the preprocessed output.
- This option lists both compiler predefined macros and user-defined macros.

Related information

- [“-E” on page 31](#)
- [“-P” on page 47](#)

-qspill

Category

[Compiler customization](#)

Pragma equivalent

None.

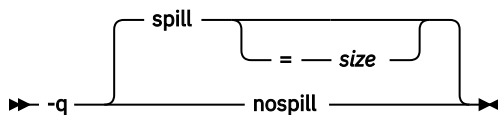
Purpose

Specifies the size (in bytes) of the register spill space, the internal program storage areas used by the optimizer for register spills to storage.

When the **-qspill** compiler option is in effect, you can specify the size of the spill area to be used for the compilation.

When the **-qnospill** compiler option is in effect, the compiler defaults to **-qspill=256**.

Syntax



Defaults

-qspill=256

Parameters

size

An integer representing the number of bytes for the register allocation spill area.

Usage

When too many registers are in use at once, the compiler saves the contents of some registers in temporary storage, called the spill area.

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.

The maximum spill area size is 1073741823 bytes or $2^{30}-1$ bytes. Typically, you will only need to specify this option when compiling very large programs with **-qoptimize**.

Note: There is an upper limit for the combined area for your spill area, local variables, and arguments passed to called functions at **-qoptimize**. For best use of the stack, do not pass large arguments, such as structures, by value.

Predefined macros

None.

Examples

If you received a warning message when compiling `myprogram.c` and want to compile it specifying a spill area of 900 entries, enter:

```
xlclang myprogram.c -qspill=900
```

-qstackprotect

Category

[Error checking and debugging](#)

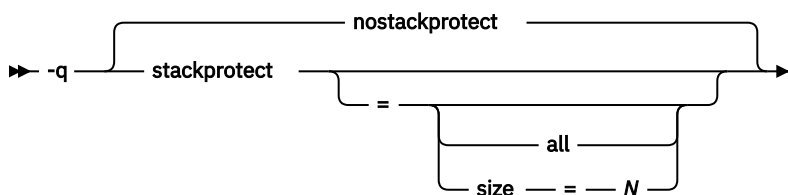
Pragma equivalent

None.

Purpose

Provides protection against malicious input data or programming errors that overwrite or corrupt the stack.

Syntax



Defaults

-qnostackprotect

Parameters

all

Protects all functions whether or not functions have vulnerable objects.

size=N

Protects all functions that contain automatic arrays whose sizes are greater than or equal to N bytes. N must be an integer value that cannot exceed $2^{31} - 2$. The default size is 8 bytes when the **-qstackprotect** option is enabled.

Usage

-qstackprotect generates extra code to protect functions with vulnerable objects against stack corruption. The **-qstackprotect** option is disabled by default because it can degrade runtime performance.

Note: The protection takes effect only if the compilation unit that contains the `main` function is compiled with **-qstackprotect**. Otherwise, the protection does not apply to any linked libraries even if the libraries have been compiled with **-qstackprotect**.

The compiler might optimize certain procedures into leaf procedures. In this case, **-qstackprotect** is not enabled for the procedure and a warning message is generated if the **-Wstack-protector** option is enabled.

Predefined macros

None.

Examples

To generate code to protect all functions, enter the following command:

```
xlclang myprogram.c -qstackprotect=all
```

To generate code to protect functions with objects of certain size, enter the following command with the size suboption set to the object size indicated in bytes:

```
xlclang myprogram.c -qstackprotect=size=8
```

-qstrict

Category

[Optimization and tuning](#)

Pragma equivalent

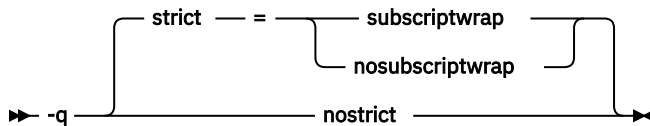
None.

Purpose

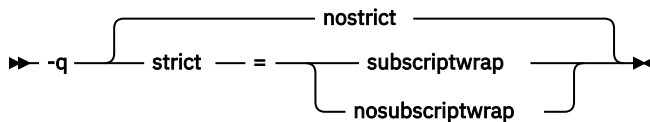
Ensures that optimizations that are done by default at the **-O3** and higher optimization levels, and, optionally at **-O2**, do not alter the semantics of a program.

Syntax

For **-qnooptimize** and **-qoptimize=2**:



For **-qoptimize=3**:



Defaults

- For **-qnoopt** and **-qopt=2**, the default option is **-qstrict**.
- For **-qopt=3**, the default option is **-qnostrict**.

Parameters

subscriptwrap | nosubscriptwrap

Prevents the compiler from assuming that array subscript expressions will never overflow.

Usage

Used to prevent optimizations done by default at optimization levels **-qopt=3**, and, optionally at **-qopt=2**, from re-ordering instructions that could introduce rounding errors.

Usage

When using **-O2** or higher optimization, if the intended truncation or sign extension of a loop induction variable resulting from variable overflow or wrap-around does not occur, you can specify **-qstrict_induction** to prevent induction variable optimizations. However, use of **-qstrict_induction** is generally not recommended because it can cause considerable performance degradation.

Predefined macros

None.

Related information

- [“-O, -qoptimize” on page 44](#)

-qsyntaxonly (-fsyntax-only)

Category

[Error checking and debugging](#)

Pragma equivalent

None.

Purpose

Performs syntax checking without generating an object file.

Syntax

►► -f — syntax-only ◄◄

►► -q — syntaxonly ◄◄

Defaults

By default, source files are compiled and linked to generate an executable file.

Usage

The **-P**, **-E**, and **-C** options override the **-fsyntax-only** (**-qsyntaxonly**) option, which in turn overrides the **-c** and **-o** options.

The **-fsyntax-only** (**-qsyntaxonly**) option suppresses only the generation of an object file. All other files, such as listing files, are still produced if their corresponding options are set.

Predefined macros

None.

Examples

To check the syntax of `myprogram.c` without generating an object file, enter:

```
xlclang myprogram.c -fsyntax-only
```

Related information

- [“-C” on page 28](#)
- [“-c” on page 29](#)
- [“-E” on page 31](#)
- [“-o” on page 46](#)
- [“-P” on page 47](#)

-qtemplatedepth (-ftemplate-depth) (C++ only)

Category

[Template control](#)

Pragma equivalent

None.

Purpose

Specifies the maximum number of recursively instantiated template specializations that will be processed by the compiler.

Syntax

►► -q *templatedepth* *—* = *— number* ►►

►► -f *template-depth* *—* = *— number* ►►

Defaults

-qtemplatedepth=300 or **-ftemplate-depth=300**

Parameters

number

The maximum number of recursive template instantiations. The number can be a value in the range of 1 to INT_MAX. If your code attempts to recursively instantiate more templates than *number*, compilation halts and an error message is issued. If you specify an invalid value, the default value of 300 is used.

Usage

Note that setting this option to a high value can potentially cause an out-of-memory error due to the complexity and amount of code generated.

Predefined macros

None.

Examples

To allow the following code in `myprogram.cpp` to be compiled successfully:

```
template <int n> void myfunc() {  
    myfunc<n-1>();  
}  
  
template <> void myfunc<0>() {}
```

```
int main() {  
    myfunc<400>();  
}
```

Enter:

```
xlclang++ myprogram.cpp -ftemplate-depth=400
```

-qthreaded

Category

[Optimization and tuning](#)

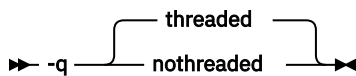
Pragma equivalent

None.

Purpose

Indicates to the compiler whether it must generate threadsafe code.

Syntax



Defaults

-qthreaded

Usage

To maintain thread safety, always specify the **-qthreaded** option when compiling or linking multithreaded applications. This option does not make code threadsafe, but it ensures that code already threadsafe remains so after compilation and linking. It also ensures that all optimizations are threadsafe.

Specifying the **-qnothreaded** option enables the optimizers to perform non-threadsafe transformations for single threaded programs.

Predefined macros

None.

-qtune

Category

[Optimization and tuning](#)

Pragma equivalent

None.

Purpose

Tunes instruction selection, scheduling, and other architecture-dependent performance enhancements to run best on a specific hardware architecture.

Syntax

►► -q — tune — = — *n* ►►

Defaults

-qtune(10)

Parameters

n

Specifies the group to which a model number belongs as a sub-parameter. If you specify a model which does not exist or is not supported, a warning message is issued stating that the suboption is invalid and that the default will be used. Current models that are supported include:

5

This option generates code that is executable on all models but that is optimized for the model 2064-100 (z900) in z/Architecture mode.

6

This option generates code that is executable on all models, but is optimized for the 2084-xxx (z990) models.

7

This option generates code that is executable on all models, but is optimized for the 2094-xxx (IBM System z9 Enterprise Class) and 2096-xxx (IBM System z9 Business Class) models.

8

This option is the default. This option generates code that is executable on all models, but is optimized for the 2097-xxx (IBM System z10 Enterprise Class) and 2098-xxx (IBM System z10 Business Class) models.

9

This option generates code that is executable on all models, but is optimized for the 2817-xxx (IBM zEnterprise 196 (z196)) and 2818-xxx (IBM zEnterprise 114 (z114)) models.

10

This option generates code that is executable on all models, but is optimized for the 2827-xxx (IBM zEnterprise EC12 (zEC12)) and 2828-xxx (IBM zEnterprise BC12 (zBC12)) models.

11

This option generates code that is executable on all models, but is optimized for the 2964-xxx (IBM z13[®] (z13)) and the 2965-xxx (IBM z13s (z13s)) models.

12

This option generates code that is executable on all models, but is optimized for the 3906-xxx (IBM z14) models.

13

This option generates code that is executable on all models, but is optimized for the 8561-xxx (IBM z15) models.

Note: For these system machine models, x indicates any value. For example, 9672-Rx4 means 9672-RA4 through to 9672-RY4 and 9672-R14 through to 9672-R94 (the entire range of G3 processors), not just 9672-RX4.

Usage

The **-qtune** option specifies the architecture for which the executable program will be optimized. The **-qtune** level controls how the compiler selects and orders the available machine instructions, while staying within the restrictions of the **-qarch** level in effect. It does so in order to provide the highest performance possible on the given **-qtune** architecture from those that are allowed in the generated code. It also controls instruction scheduling (the order in which instructions are generated to perform a

particular operation). Note that **-qtune** impacts performance only; it does not impact the processor model on which you will be able to run your application.

Select **-qtune** to match the architecture of the machine where your application will run most often. Use **-qtune** in cooperation with **-qarch**. **-qtune** must always be greater or equal to **-qarch** because you will want to tune an application for a machine on which it can run. The compiler enforces this by adjusting **-qtune** up rather than **-qarch** down. **-qtune** does not specify where an application can run. It is primarily an optimization option. For many models, the best **-qtune** level is not the best **-qarch** level. For example, the correct choices for model 9672-Rx5 (G4) are **-qarch=2** and **-qtune=3**.

Note: If the **-qtune** level is lower than the specified **-qarch** level, the compiler forces **-qtune** to match the **-qarch** level or uses the default **-qtune** level, whichever is greater.

Predefined macros

None.

Related information

- [“-qarch” on page 54](#)

Supported GCC options

The following GCC options are also supported in XL C/C++ V2.4.1. For details about these options, see GNU Compiler Collection online documentation(<http://gcc.gnu.org/onlinedocs/>).

Table 21. Supported GCC options	
GCC option	-q option synonym
-faccess-control	
-fasm, -fno-asm	-qasm
-fbracket-depth	
-fcolor-diagnostics	
-fconstexpr-depth	
-fdiagnostics-fixit-info	
-fdiagnostics-format=[clang msvc vi]	
-fdiagnostics-print-source-range-info	
-fdiagnostics-show-category=[none id name]	
-fdiagnostics-show-option	
-felide-type	
-fexec-charset	
-ffreestanding	
-fhosted	
-fgnu-keywords	
-fgnu89-inline	
-fmessage-length	
-foperator-names	

<i>Table 21. Supported GCC options (continued)</i>	
GCC option	-q option synonym
-frtti	-qrtti
-fsigned-char	-qchars=signed
-fshow-column	
-fshow-source-location	
-fsyntax-only	-qsyntaxonly
-ftemplate-backtrace-limit	
-ftemplate-depth	-qtemplatedepth
-funsigned-char	-qchars=unsigned
-isystem	
-pedantic	
-W	

Chapter 4. Compiler pragmas reference

The following sections describe the available pragmas:

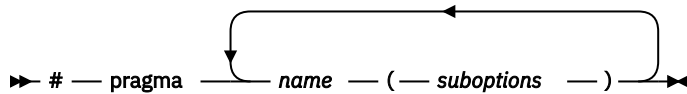
- “Pragma directive syntax” on page 101
- “Scope of pragma directives” on page 101
- “Supported IBM pragmas” on page 102

Pragma directive syntax

XL C/C++ V2.4.1 supports the following forms of pragma directives:

#pragma *name*

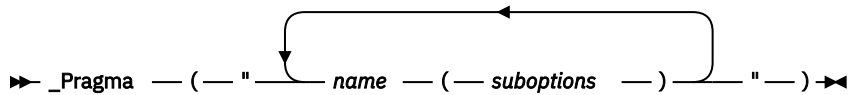
This form uses the following syntax:



The *name* is the pragma directive name, and the *suboptions* are any required or optional suboptions that can be specified for the pragma, where applicable.

_Pragma ("*name*")

This form uses the following syntax:



For example, the statement:

```
_Pragma ( "pack(1)" )
```

is equivalent to:

```
#pragma pack(1)
```

For all forms of pragma statements, you can specify more than one *name* and *suboptions* in a single **#pragma** statement.

The *name* on a pragma is subject to macro substitutions, unless otherwise stated. The compiler ignores unrecognized pragmas, issuing an informational message indicating this.

Scope of pragma directives

Many pragma directives can be specified at any point within the source code in a compilation unit; others must be specified before any other directives or source code statements. In the individual descriptions for each pragma, the "Usage" section describes any constraints on the pragma's placement.

In general, if you specify a pragma directive before any code in your source program, it applies to the entire compilation unit, including any header files that are included. For a directive that can appear anywhere in your source code, it applies from the point at which it is specified, until the end of the compilation unit.

You can further restrict the scope of a pragma's application by using complementary pairs of pragma directives around a selected section of code.

Many pragmas provide pop or reset suboptions that allow you to enable and disable pragma settings in a stack-based fashion; examples of these are provided in the relevant pragma descriptions.

Supported IBM pragmas

This section contains descriptions of individual pragmas available in XL C/C++ V2.4.1.

For each pragma, the following information is given:

Purpose

This section provides a brief description of the effect of the pragma, and why you might want to use it.

Syntax

This section provides the syntax for the pragma. For convenience, the **#pragma name** form of the directive is used in each case. However, it is perfectly valid to use the alternate C99-style `_Pragma` operator syntax; see [“Pragma directive syntax” on page 101](#) for details.

Parameters

This section describes the suboptions that are available for the pragma, where applicable.

Usage

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

Examples

Where appropriate, examples of pragma directive use are provided in this section.

#pragma convert

Purpose

Provides a way to specify more than one coded character set in a single compilation unit to convert string literals.

Unlike the **-qascii** compiler option, it allows for more than one character encoding to be used for string literals in the same compilation unit.

Syntax

➤ # — pragma — convert — (— " — *code_set_name* — " —) ➤
pop

Parameters

code_set_name

Is a string that specifies an ASCII or EBCDIC based codepage that is not DBCS or UTF-8.

pop

Resets the code set to that which was previously in effect immediately before the current codepage.

Usage

The compiler option **-qascii** determines the code set in effect before any `#pragma convert` directives are introduced, and after all `#pragma convert` directives are popped from the stack.

The conversion continues from the point of placement of the first `#pragma convert` directive until another `#pragma convert` directive is encountered, or until the end of the main source file. For every `#pragma convert` directive in your program, it is good practice to have a corresponding `#pragma convert(pop)` as well. This will prevent one file from potentially changing the codepage of another file that is included.

The following are not converted:

- A string or character constant specified in hexadecimal or octal escape sequence format (because it represents the *value* of the desired character on output).
- A string literal that is part of a `#include` or `pragma` directive.
- **C++** String literals that are used to specify linkage (for example, `extern "C"`). **C++**
- A wide character string or wide character literals.
- A string in an `asm` statement.

Related information

- [“-qascii” on page 55](#)

#pragma csect

Purpose

Identifies the name for the code or static control section (CSECT) of the object module.

Syntax

➤ # — **pragma** — **csect** — (— **code** — , — " — *name* — " —) ➤
 └── **static** ─┘

Parameters

code

Specifies the CSECT that contains the executable code (C functions) and constant data.

static

Designates the CSECT that contains all program variables with the `static` storage class and all character strings.

name

The name that is used for the applicable CSECT. The compiler does not map the name in any way. The name must not conflict with the name of an exposed name (external function or object) in a source file. In addition, it must not conflict with another **#pragma csect** directive or **#pragma map** directive. For example, the name of the code CSECT must differ from the name of the static CSECTs.

Usage

At most, two **#pragma csect** directives can appear in a source program as follows:

- One for the code CSECT
- One for the static CSECT

When both **#pragma csect** and the CSECT compiler option are specified, the compiler first uses the option to generate the CSECT names, and then the **#pragma csect** overrides the names generated by the option.

Examples

Suppose that you compile the following code with the option `CSECT(abc)` and program name `foo.c`.

```
#pragma csect (STATIC, "blah")
int main ()
{
    return 0;
}
```

First, the compiler generates the following CSECT names:

```
STATIC: abc#foo#S  
CODE: abc#foo#C
```

Then the **#pragma csect** overrides the static CSECT name, which renders the final CSECT name to be:

```
STATIC: blah  
CODE: abc#foo#C
```

Related information

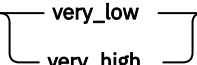
- [“-qcsect” on page 61](#)

#pragma execution_frequency

Purpose

Marks program source code that you expect will be either very frequently or very infrequently executed. When optimization is enabled, the pragma is used as a hint to the optimizer.

Syntax

➤ # — pragma — execution_frequency — (—  —) ➤

Parameters

very_low

Marks source code that you expect will be executed very infrequently.

very_high

Marks source code that you expect will be executed very frequently.

Usage

Use this pragma in conjunction with an optimization option; if optimization is not enabled, the pragma has no effect.

The pragma must be placed within block scope, and acts on the closest preceding point of branching.

Examples

In the following example, the pragma is used in an `if` statement block to mark code that is executed infrequently.

```
int *array = (int *) malloc(10000);  
  
if (array == NULL) {  
    /* Block A */  
    #pragma execution_frequency(very_low)  
    error();  
}
```

In the next example, the code block Block B is marked as infrequently executed and Block C is likely to be chosen during branching.

```
if (Foo > 0) {  
    #pragma execution_frequency(very_low)  
    /* Block B */  
    doSomething();  
} else {  
    /* Block C */  
}
```

```
doAnotherThing();
}
```

In this example, the pragma is used in a switch statement block to mark code that is executed frequently.

```
while (counter > 0) {
    #pragma execution_frequency(very_high)
    doSomething();
} /* This loop is very likely to be executed. */

switch (a) {
    case 1:
        doOneThing();
        break;
    case 2:
        #pragma execution_frequency(very_high)
        doTwoThings();
        break;
    default:
        doNothing();
} /* The second case is frequently chosen. */
```

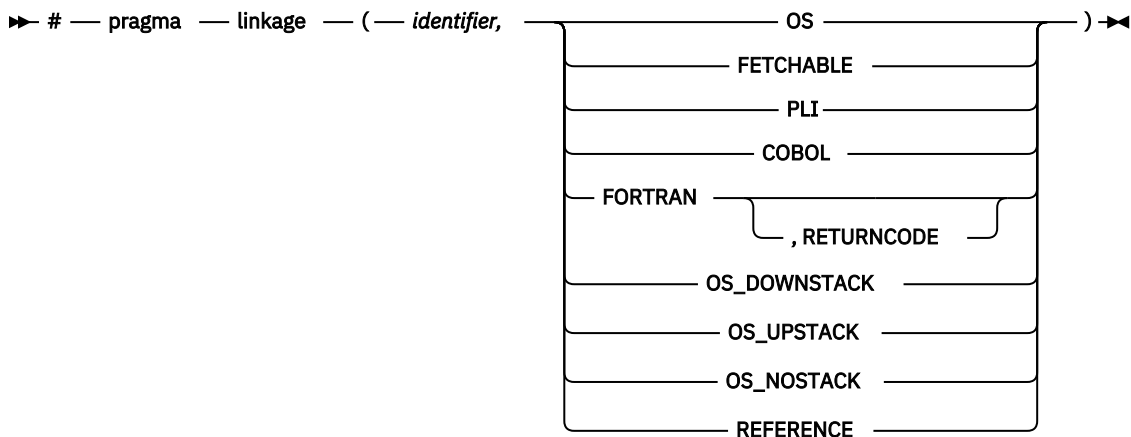
#pragma linkage (C only)

Purpose

Identifies the entry point of modules that are used in interlanguage calls from C programs as well as the linkage or calling convention that is used on a function call.

The directive also designates other entry points within a program that you can use in a `fetch` operation.

Syntax



Defaults

XPLINK linkage.

Parameters

identifier

The name of a function that is to be the entry point of the module, or a typedef name that will be used to define the entry point. (See below for an example.)

FETCHABLE

Indicates that *identifier* can be used in a `fetch` operation. A fetched XPLINK module must have its entry point defined with a **#pragma linkage(..., fetchable)** directive.

OS

Designates *identifier* as an OS linkage entry point. OS linkage is the basic linkage convention that is used by the operating system.

PLI

Designates *identifier* as a PL/I linkage entry point.

COBOL

Designates *identifier* as a COBOL linkage entry point.

FORTTRAN

Designates *identifier* as a FORTRAN linkage entry point.

RETURNCODE indicates to the compiler that the routine named by *identifier* is a FORTRAN routine, which returns an alternate return code. It also indicates that the routine is defined outside the translation unit. You can retrieve the return code by using the `fortrc` function. If the compiler finds the function definition inside the translation unit, it issues an error message. Note that you can define functions outside the translation unit, even if you do not specify the RETURNCODE keyword.

OS_DOWNSTACK

Designates *identifier* as an OS linkage entry point in XPLINK mode with a downward growing stack frame.

OS_UPSTACK

Designates *identifier* as an OS linkage entry point in XPLINK mode with a traditional upward growing stack frame.

This linkage is required for a new XPLINK downward-stack routine to be able to call a traditional upward-stack OS routine. This linkage explicitly invokes compatibility code to swap the stack between the calling and the called routines.

OS_NOSTACK

Designates *identifier* as an OS linkage entry point in XPLINK mode with no preallocated stack frame. An argument list is constructed containing the addresses of the actual arguments. The size of the address is 4-byte for 31-bit and 8-byte for 64-bit. Register 1 is set to point to this argument list. For 31-bit only, the last item in this list has its high order bit set. For integer type arguments, the value passed is widened to the size of the `int` type, that is 4-byte. Register 13 points to a save area that may not be followed by z/OS Language Environment control structures, such as the NAB. The size of the save area is 72-byte for 31-bit and 144-byte for 64-bit. Register 14 contains the return address. Register 15 contains the entry point of the called function.

REFERENCE

This is synonymous with `OS_DOWNSTACK` in XPLINK mode. Unlike the linkage `OS`, this is not affected by the `OSCALL` suboption of XPLINK.

Usage

You can use a typedef in a `#pragma linkage` directive to associate a specific linkage convention with a function type. In the following example, the directive associates the OS linkage convention with the typedef `func_t`:

```
typedef void func_t(void);
#pragma linkage (func_t,OS)
```

This typedef can then be used in C declarations wherever a function should have OS linkage. In the following example:

```
func_t myfunction;
```

`myfunction` is declared as having type `func_t`, which is associated with OS linkage; `myfunction` would therefore have OS linkage.

#pragma leaves

Purpose

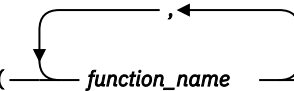
Informs the compiler that a named function never returns to the instruction following a call to that function.

By informing the compiler that it can ignore any code after the function, the directive allows for additional opportunities for optimization.

This pragma is commonly used for custom error-handling functions, in which programs can be terminated if a certain error is encountered.

Syntax

► # — pragma — leaves — (— *function_name* —) ►



Parameters

function_name

The name of the function that does not return to the instruction following the call to it.

Defaults

Not applicable.

Usage

If you specify the **-qlibansi** compiler option (which informs the compiler that function names that match functions in the C standard library are in fact C library functions), the compiler checks whether the `longjmp` family of functions (`longjmp`, `_longjmp`, `siglongjmp`, and `_siglongjmp`) contain `#pragma leaves`. If the functions do not contain this pragma directive, the compiler will insert this directive for the functions. This is not shown in the listing.

Examples

```
#pragma leaves(handle_error_and_quit)
void test_value(int value)
{
    if (value == ERROR_VALUE){
        handle_error_and_quit(value);
        TryAgain();    // optimizer ignores this because
                       // never returns to execute it
    }
}
```

Related information

- [“#pragma reachable” on page 112.](#)

#pragma map

Purpose

Converts all references to an identifier to another, externally defined identifier.

Syntax

#pragma map syntax (C only)





➤ # — pragma — map — (— *name1* — , — " — *name2* — " —) ➤

#pragma map syntax (C++ only)

➤ # — pragma — map — (— *name1* — (— *argument_list* —) — , — " — *name2* — " —) ➤


Parameters

name1

The name used in the source code.  *name1* can represent a data object or function with external linkage.   *name1* can represent a data object, a non-overloaded or overloaded function, or overloaded operator, with external linkage.  If the name to be mapped is not in the global namespace, it must be fully qualified.


name1 should be declared in the same compilation unit in which it is referenced, but should not be defined in any other compilation unit. *name1* must not be used in another **#pragma map** directive or any assembly label declaration anywhere in the program.

argument_list

The list of arguments for the overloaded function or operator function designated by *name1*. If *name1* designates an overloaded function, the function must be parenthesized and must include its argument list if it exists. If *name1* designates a non-overloaded function, only *name1* is required, and the parentheses and argument list are optional. 

name2

The name that will appear in the object code.  *name2* can represent a data object or function with external linkage.

 *name2* can represent a data object, a non-overloaded or overloaded function, or overloaded operator, with external linkage. *name2* must be specified using its mangled name. To obtain C++ mangled names, compile your source to object files only, using the **-c** compiler option, and use the **nm** operating system command on the resulting object file.

If the name exceeds 65535 bytes, an informational message is emitted and the pragma is ignored.

name2 may or may not be declared in the same compilation unit in which *name1* is referenced, but must not be defined in the same compilation unit. Also, *name2* should not be referenced anywhere in the compilation unit where *name1* is referenced. *name2* must not be the same as that used in another **#pragma map** directive or any assembly label declaration in the same compilation unit.

Usage

The **#pragma map** directive can appear anywhere in the program. Note that in order for a function to be actually mapped, the map target function (*name2*) must have a definition available at link time (from another compilation unit), and the map source function (*name1*) must be called in your program.

You cannot use **#pragma map** with compiler built-in functions.

Examples

The following is an example of **#pragma map** used to map a function name (using the mangled name for the map name in C++):

```
/* Compilation unit 1: */  
  
#include <stdio.h>  
  
void func();  
extern void bar(); /* optional */  
  
#if __cplusplus
```



```

#pragma map (func, "_Z3barv")
#else
#pragma map (func, "bar")
#endif
int main()
{
func();
}

/* Compilation unit 2: */

#include <stdio.h>

void bar()
{
printf("Hello from func bar!\n");
}

```

The call to `func` in compilation unit 1 resolves to a call to `bar`:

```
Hello from func bar!
```

C++ The following is an example of **#pragma map** used to map an overloaded function name (using C linkage, to avoid using the mangled name for the map name):

```

// Compilation unit 1:

#include <iostream>
#include <string>

using namespace std;

void func();
void func(const string&);
extern "C" void bar(const string&); // optional

#pragma map (func(const string&), "bar")

int main()
{
func("Have a nice day!");
}

// Compilation unit 2:

#include <iostream>
#include <string>

using namespace std;

extern "C" void bar(const string& s)
{
cout << "Hello from func bar!" << endl;
cout << s << endl;
}

```

The call to `func(const string&)` in compilation unit 1 resolves to a call to `bar(const string&)`:

```
Hello from func bar!
Have a nice day!
```

#pragma option_override

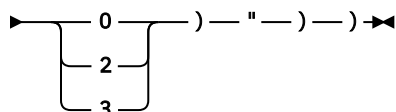
Purpose

Allows you to specify optimization options at the subprogram level that override optimization options given on the command line.

This enables finer control of program optimization, and can help debug errors that occur only under optimization.

Syntax

➤ # — pragma — option_override — (— *identifier* — , — " — opt — (— level — , — ➤



Parameters

identifier

The name of a function for which optimization options are to be overridden.

The following table shows the equivalent command line option for each pragma suboption.

#pragma option_override value	Equivalent compiler option
level, 0	-O
level, 2	-O2
level, 3	-O3

Usage

The pragma takes effect only if optimization is already enabled by a command-line option. You can only specify an optimization level in the pragma *lower* than the level applied to the rest of the program being compiled.

The **#pragma option_override** directive only affects functions that are defined in the same compilation unit. The pragma directive can appear anywhere in the translation unit. That is, it can appear before or after the function definition, before or after the function declaration, before or after the function has been referenced, and inside or outside the function definition.

C++ This pragma cannot be used with overloaded member functions.

Examples

Suppose you compile the following code fragment containing the functions myfunc1 and myfunc2 using **-O2**. Because the code contains the `#pragma option_override(myfunc2, "opt(level, 0)")` directive, function myfunc2 will not be optimized.

```
myfunc1(){
    .
    .
}
#pragma option_override(myfunc2, "opt(level, 0)")
myfunc2(){
    .
    .
}
```

Related information

- [“-O, -qoptimize” on page 44](#)

#pragma priority (C++ only)

Purpose

Specifies the priority level for the initialization of static objects.

The C++ standard requires that all global objects within the same translation unit be constructed from top to bottom, but it does not impose an ordering for objects declared in different translation units.

Syntax

pragma syntax

➤ # — pragma — priority — (— *number* —) ➤

Defaults

The default priority level is 0.

Parameters

number

An integer literal in the range of -2147482624 to 2147483647. A lower value indicates a higher priority; a higher value indicates a lower priority. Numbers from -2147483648 to -2147482623 are reserved for system use. If you do not specify a *number*, the compiler assumes 0.

Usage

In order to be consistent with the Standard, priority values specified within the same translation unit must be strictly increasing. Objects with the same priority value are constructed in declaration order.

Note: The C++ variable attribute `init_priority` can also be used to assign a priority level to a shared variable of class type.

Examples

To compile the file `myprogram.C` to produce an object file `myprogram.o` so that objects within that file have an initialization priority of 2000, enter the following command:

```
xlclang++ myprogram.C -c -qpriority=2000
```

#pragma weak (C only)

Purpose

Prevents the binder from issuing error messages if it encounters a symbol multiply-defined during linking, or if it does not find a definition for a symbol.

The pragma can be used to allow a program to call a user-defined function that has the same name as a library function. By marking the library function definition as "weak", the programmer can reference a "strong" version of the function and cause the binder to accept multiple definitions of a global symbol in the object code. While this pragma is intended for use primarily with functions, it will also work for most data objects.

Syntax

#pragma weak

➤ # — pragma — weak — *name1* ➤

Parameters

name1

A name of a data object or function with external linkage.

Usage

There are two forms of the **weak** pragma:

#pragma weak *name1*

This form of the pragma marks the definition of the *name1* as "weak" in a given compilation unit. If *name1* is referenced from anywhere in the program, the binder will use the "strong" version of the definition (that is, the definition not marked with `#pragma weak`), if there is one. If there is no strong definition, the binder will use the weak definition; if there are multiple weak definitions, it is unspecified which weak definition the binder will select (typically, it uses the definition found in the first object file specified on the command line during the link step). *name1* must be defined in the same compilation unit as `#pragma weak`.

Note: This pragma should not be used with uninitialized global data.

Example

The following is an example of the `#pragma weak name1` form:

```
// Compilation unit 1:
#include <stdio.h>

void func();

int main(){
    func();
}

// Compilation unit 2:
#include <stdio.h>

#ifdef __cplusplus
#pragma weak func
#endif

void func(){
    printf("func called from compilation unit 2\n");
}

// Compilation unit 3:
#include <stdio.h>

void func(){
    printf("func called from compilation unit 3\n");
}
```

If all three compilation units are compiled and linked together, the binder will use the strong definition of `func` in compilation unit 3 for the call to `func` in compilation unit 1, and the output will be:

```
func called from compilation unit 3
```

If only compilation unit 1 and 2 are compiled and linked together, the binder will use the weak definition of `func` in compilation unit 2, and the output will be:

```
func called from compilation unit 2
```

#pragma reachable

Purpose

Informs the compiler that the point in the program after a named function can be the target of a branch from some unknown location.

By informing the compiler that the instruction after the specified function can be reached from a point in your program other than the return statement in the named function, the pragma allows for additional opportunities for optimization. The compiler automatically inserts `#pragma reachable` directives for the `setjmp` family of functions (`setjmp`, `_setjmp`, `sigsetjmp`, and `_sigsetjmp`) when you include the `setjmp.h` header file.

Syntax

➤ # — pragma — reachable — (— *function_name* —) ➤

Parameters

function_name

The name of a function preceding the instruction which is reachable from a point in the program other than the function's return statement.

Defaults

Not applicable.

Usage

Unlike the `#pragma leaves`, `#pragma reachable` is required by the compiler optimizer whenever the instruction following the call may receive control from some program point other than the return statement of the called function. If this condition is true and `#pragma reachable` is not specified, then the subprogram containing the call should not be compiled with **-0**, **-02**, or **-03**.

If you specify the **-qlibansi** compiler option (which informs the compiler that function names that match functions in the C standard library are in fact C library functions), the compiler checks whether the `setjmp` family of functions (`setjmp`, `_setjmp`, `sigsetjmp`, and `_sigsetjmp`) contain `#pragma reachable`. If the functions do not contain this pragma directive, the compiler will insert this directive for the functions. This is not shown in the listing.

Chapter 5. Compiler predefined macros

The compiler provides many other macros that you can use in your programs. For example, XL C/C++ V2.4.1 supports function-like macros with a variable number of arguments, as a language extension for compatibility with C and as part of C++11.

These macros have compiler-predefined values. These predefined macros are used to conditionally compile code for specific compilers, specific versions of compilers, specific environments, and specific language features. Some predefined macros are *protected*, which means that the compiler will issue a warning message if you try to undefine or redefine them. Some predefined macros are unprotected and can be undefined or redefined without warning.

Predefined macros fall into several categories:

- “General macros” on page 115
- “Macros related to the platform” on page 116
- “Macros related to compiler features” on page 117

General macros

The following predefined macros are always predefined by the compiler. Unless noted otherwise, all the following macros are *protected*, which means that the compiler will issue a warning if you try to undefine or redefine them.

Table 22. General predefined macros		
Predefined macro name	Description	Predefined value
<code>__BASE_FILE__</code>	Indicates the name of the primary source file.	The fully qualified file name of the primary source file.
<code>__DATE__</code>	Indicates the date that the source file was preprocessed.	A character string containing the date when the source file was preprocessed.
<code>__FILE__</code>	Indicates the name of the preprocessed source file.	A character string containing the name of the preprocessed source file.
<code>__FUNCTION__</code>	Indicates the name of the function currently being compiled.	A character string containing the name of the function currently being compiled.
<code>__LINE__</code>	Indicates the current line number in the source file.	An integer constant containing the line number in the source file.
<code>__SIZE_TYPE__</code>	Indicates the underlying type of <code>size_t</code> on the current platform. Not protected.	<code>unsigned long</code>
<code>__TIME__</code>	Indicates the time that the source file was preprocessed.	A character string containing the time when the source file was preprocessed.

Table 22. General predefined macros (continued)

Predefined macro name	Description	Predefined value
__TIMESTAMP__	Indicates the date and time when the source file was last modified. The value changes as the compiler processes any include files that are part of your source program.	<p>A character string literal in the form "Day Mmm dd hh:mm:ss yyyy", where:</p> <p>Day Represents the day of the week (Mon, Tue, Wed, Thu, Fri, Sat, or Sun).</p> <p>Mmm Represents the month in an abbreviated form (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec).</p> <p>dd Represents the day. If the day is less than 10, the first d is a blank character.</p> <p>hh Represents the hour.</p> <p>mm Represents the minutes.</p> <p>ss Represents the seconds.</p> <p>yyyy Represents the year.</p>

Macros indicating the z/OS XL C/C++ compiler

Most of the macros related to the XL C/C++ V2.4.1 compiler are predefined and protected, which means that the compiler will issue a warning if you try to undefine or redefine them. You can use the **-E** compiler option to view the values of the predefined macros. You can use these macros to distinguish code consumable by XL C/C++ V2.4.1 from code consumed by other compilers in your programs.

Macros related to the platform

The following predefined macros are provided to facilitate porting applications between platforms. All platform-related predefined macros are unprotected and can be undefined or redefined without warning unless otherwise specified.

Table 23. Platform-related predefined macros

Predefined macro name	Description	Predefined value	Predefined under the following conditions
__370__	Indicates that the program is compiled or targeted to run on IBM System/370.	1	Always predefined.
__HHW_370__	Indicates that the host hardware is System/370.	1	Always predefined for z/OS.

Table 23. Platform-related predefined macros (continued)			
Predefined macro name	Description	Predefined value	Predefined under the following conditions
__HOS_MVS__	Indicates that the host operating system is z/OS.	1	Always predefined for z/OS.
__MVS__	Indicates that the host operating system is z/OS.	1	Always predefined for z/OS.
__THW_370__	Indicates that the target hardware is System/370.	1	Always predefined for z/OS
__TOS_MVS__	Indicates that the host operating system is z/OS.	1	Always predefined for z/OS.

Macros related to compiler features

Feature-related macros are predefined according to the setting of specific compiler options or pragmas. Unless noted otherwise, all feature-related macros are protected, which means that the compiler will issue a warning if you try to undefine or redefine them.

Feature-related macros are discussed in the following sections:





- “[Macros related to compiler option settings](#)” on page 117
- “[Macros related to language levels](#)” on page 118

Macros related to compiler option settings

The following macros can be tested for various features, including source input characteristics, output file characteristics, and optimization. All of these macros are predefined by a specific compiler option or suboption, or any invocation or pragma that implies that suboption. If the suboption enabling the feature is not in effect, then the macro is undefined.

Table 24. General option-related predefined macros			
Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect
__ARCH__	Indicates the target architecture for which the source code is being compiled.	The integer value specified in the “ -qarch ” on page 54 compiler option.	-qarch=level
__BFP__	Indicates that binary floating point (BFP) mode is in effect.	1	Always defined.
_CHAR_SIGNED __CHAR_SIGNED__	Indicates that the default character type is signed char.	1	-qchars=signed (-fsigned-char)
_CHAR_UNSIGNED __CHAR_UNSIGNED__	Indicates that the default character type is unsigned char.	1	-qchars=unsigned (-funsigned-char)
__CHARSET_LIB	Indicates support for processing ASCII data natively at execution time.	1	“-qascii” on page 55

Table 24. General option-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect
<code>__IBM_UTF_LITERAL</code>	Indicates support for UTF-16 and UTF-32 string literals.	1	 <code>-qlanglvl=extc89</code>  <code>-qlanglvl=extended0x</code>
<code>__IGNERRNO__</code>	Indicates that system calls do not modify <code>errno</code> , thereby enabling certain compiler optimizations.	1	<code>-qignerrno</code>
<code>__LIBANSI__</code>	Indicates that calls to functions whose names match those in the C Standard Library are in fact the C library functions, enabling certain compiler optimizations.	1	<code>-qlibansi</code>
<code>__OPTIMIZE__</code>	Indicates the level of optimization in effect.	The integer value specified in the “ <code>-O</code> , <code>-qoptimize</code> ” on page 44 compiler option.	“ <code>-O</code> , <code>-qoptimize</code> ” on page 44
<code>__OPTIMIZE_SIZE__</code>	Indicates that optimization for code size is in effect.	1	<code>-O</code> <code>-O2</code> <code>-O3</code> and “ <code>-qcompact</code> ” on page 60
 <code>__RTTI_ALL__</code>	Generates runtime type identification (RTTI) information for classes with virtual functions.	1	“ <code>-qrtti</code> (<code>-frtti</code>) (C++ only)” on page 87
 <code>__NO_RTTI__</code>	Indicates that no runtime type identification (RTTI) information is generated.	1	<code>-qnortti</code>
<code>__TUNE__</code>	Indicates the architecture for which the compiler generated executable code is optimized.	The integer value specified in the “ <code>-qtune</code> ” on page 96 compiler option.	“ <code>-qtune</code> ” on page 96

Macros related to language levels





The following macros except  `__cplusplus`  and  `__STDC_VERSION__`  are predefined to a value of 1 by a specific language level, represented by a suboption of the “`-qlanglvl` (`-std`)” on page 75 compiler option, or any invocation or pragma that implies that suboption. If the suboption enabling the feature is not in effect, then the macro is undefined.

Table 25. Predefined macros for language levels






Predefined macro name	Description	Predefined when the following language level is in effect
 <code>__BOOL__</code>	Indicates that the <code>bool</code> keyword is accepted.	Always defined.

Table 25. Predefined macros for language levels (continued)		
Predefined macro name	Description	Predefined when the following language level is in effect
 <code>__cplusplus</code>	The numeric value that indicates the supported language standard as defined by that specific standard.	The format is <code>yyyymmL</code> . (For example, the format is <code>201103L</code> for <code>c++11</code> .)
 <code>__IBMC_GENERIC</code>	Indicates support for the generic features of C11 standard.	<code>gnu89</code> <code>gnu99</code> <code>gnu11</code> <code>c11</code>
 <code>__IBMCPP_COMPLEX_INIT</code>	Indicates support for the initialization of complex types: <code>float _Complex</code> , <code>double _Complex</code> , and <code>long double _Complex</code> .	<code>extended0x</code>
<code>__STDC__</code>	Indicates that the compiler conforms to the ANSI/ISO C or C++ standard.	Predefined to 1 if ANSI/ISO C or C++ standard conformance is in effect.
 <code>__STDC_VERSION__</code>	Indicates the version of ANSI/ISO C standard which the compiler conforms to.	The format is <code>yyyymmL</code> . (For example, the format is <code>199901L</code> for C99.)

Note: When you compile your source code with **xlcclang++** and want to use the C99 complex types that are defined in the C99 header file `complex.h`, you need to include `complex.h` and define the `__C99_COMPLEX_HEADER__` macro ; otherwise, the C++ language header file `complex` is used.

Notices

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

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

IBM Corporation
J74/G4
555 Bailey Avenue
San Jose, CA 95141-1099
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:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
3-2-12, Roppongi, Minato-ku, Tokyo 106-8711

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 publication 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.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at [Copyright and Trademark information \(www.ibm.com/legal/copytrade.shtml\)](http://www.ibm.com/legal/copytrade.shtml).

Adobe, Acrobat, PostScript and all Adobe-based trademarks are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

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

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

Standards

The following standards are supported:

- The C language is consistent with *Programming languages - C ISO/IEC 9899:1999 (C99)* and a subset of *Programming languages - C ISO/IEC 9899:2011 (C11)*. For more information, see [International Organization for Standardization \(ISO\) \(www.iso.org\)](http://www.iso.org).
- The C++ language is consistent with *Programming languages - C++ ISO/IEC 14882:1998 (C++98)*, *Programming languages - C++ ISO/IEC 14882:2003(E) (C++03)*, a subset of *Programming languages - C++ ISO/IEC 14882:2011 (C++11)*, and a subset of *Programming languages - C++ (ISO/IEC 14882:2014) (C++14)*.

The following standards are supported in combination with the z/OS UNIX System Services element:

- A subset of *IEEE Std. 1003.1-2001 (Single UNIX Specification, Version 3)*. For more information, see [IEEE \(www.ieee.org\)](http://www.ieee.org).
- *IEEE Std 1003.1—1990, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE P1003.1a Draft 6 July 1991, Draft Revision to Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*, copyright 1992 by the Institute of Electrical and Electronic Engineers, Inc.
- *IEEE Std 1003.2—1992, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shells and Utilities*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE Std P1003.4a/D6—1992, IEEE Draft Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE 754-1985 (R1990) IEEE Standard for Binary Floating-Point Arithmetic (ANSI)*, copyright 1985 by the Institute of Electrical and Electronic Engineers, Inc.
- *X/Open CAE Specification, System Interfaces and Headers, Issue 4 Version 2*, copyright 1994 by The Open Group
- *X/Open CAE Specification, Networking Services, Issue 4*, copyright 1994 by The Open Group
- United States Government's *Federal Information Processing Standard (FIPS) publication for the programming language C, FIPS-160*, issued by National Institute of Standards and Technology, 1991

Index

B

binder
 invoking [7](#)

C

CCSID (coded character set identifier) [102](#)
COBOL linkage [106](#)
command
 syntax diagrams [viii](#)
compiler generated information
 listing [10](#)
 overview [8](#)
compiler options
 by functional category [17](#)
 defaults [11](#)
 individual options [26](#)
 options overview [17](#)
 resolving conflicts [5](#)
 specifying
 command lines [4](#)
 configuration files [4](#)
 source files [4](#)
compiler options category
 customization [25](#)
 Error checking and debugging [20](#)
 floating-point [20](#)
 input control [18](#)
 language element control [18](#)
 object code control [20](#)
 optimization and tuning [24](#)
 output control [17](#)
 portability and migration [25](#)
 template control [19](#)
compiler predefined macros
 overview [115](#)
compiler reference
 pragmas [101](#)
configuration
 custom configuration files [11](#)
configuration files
 custom [12](#)
 customizing [14](#)
 editing [14](#)
convert pragma [102](#)
csect pragma [103](#)

D

debugging
 errors [86](#)
 SERVICE compiler option [88](#)
default configuration file
 example [13](#)

E

entry point
 linkage [105](#)
environment variables
 compile time [11](#)
 link time [11](#)
 runtime [11](#)
 setting [11](#)
external
 variables
 exporting [68](#)
 importing [68](#)

F

FETCHABLE preprocessor directive [105](#)
FORTRAN linkage [106](#)
functions
 exporting [68](#)
 importing [68](#)

G

GCC
 options
 summary [98](#)

I

invocation
 command syntax [1](#)
 compiling [1](#)
 input files [2](#)
 invoking [1](#)
 linking [1](#)
 output files [3](#)
 preprocessing [5](#)

L

language standards
 options [75](#)
linkage
 COBOL [106](#)
 FORTRAN [106](#)
 language [105](#)
 PL/I [106](#)
linkage pragma [105](#)
linking
 control [25](#)
 orders [8](#)
listing
 files [78](#)
 listings and messages control [23](#)

M

macros
 compiler feature [117](#)
 compiler options setting [117](#)
 language features [118](#)
 platform [116](#)
 predefined
 identify compiler [116](#)
mainframe
 education [x](#)
memory
 files, compiler work-files [81](#)

O

OS linkage [106](#)

P

PDF documents [x](#)
PL/I linkage [106](#)
pragma directives
 introduction [102](#)
 scope [101](#)
 syntax [101](#)
pragmas
 convert [102](#)
 csect [103](#)
 execution_frequency [104](#)
 linkage [105](#)
 option_override [109](#)
 reachable [112](#)
preprocessing
 directory search sequence [6](#)
programming errors [86](#)

S

sending
 feedback [xi](#)
 reader comments [xi](#)
syntax
 invocation command [1](#)
syntax diagrams
 how to read [viii](#)

T

technical support [xi](#)
typographical conventions [vii](#)

Z

z/OS Basic Skills Knowledge Center [x](#)



Product Number: 5650-ZOS

SC31-5801-00

