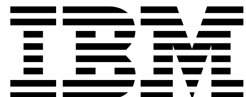
IBM XL C/C++ for Linux on z Systems, V1.2

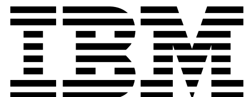


Compiler Reference

Version 1.2

SC27-5998-01

IBM XL C/C++ for Linux on z Systems, V1.2



Compiler Reference

Version 1.2

SC27-5998-01

Note

Before using this information and the product it supports, read the information in "Notices" on page 213.

First edition

This edition applies to IBM XL C/C++ for Linux on z Systems, V1.2 (Program 5725-N01) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

© Copyright IBM Corporation 2015.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this document v
Who should read this document v
How to use this document v
How this document is organized v
Conventions vi
Related information ix
IBM XL C/C++ information ix
Standards and specifications x
Other information xi
Technical support xi
How to send your comments xi

Chapter 1. Compiling and linking applications 1
Invoking the compiler 1
Command-line syntax 2
Types of input files 3
Types of output files 4
Specifying compiler options 5
Specifying compiler options on the command line 5
Specifying compiler options in a configuration file 5
Specifying compiler options in program source files 6
Resolving conflicting compiler options 6
Specifying compiler options for architecture-specific compilation 7
Preprocessing 8
Directory search sequence for included files . . . 9
Linking 10
Order of linking 11
Redistributable libraries 12
Compiler messages and listings 12
Compiler messages 13
Compiler listings 13
Paging space errors during compilation 15

Chapter 2. Configuring compiler defaults 17
Setting environment variables 17
Compile-time and link-time environment variables 18
Runtime environment variables 18
Using custom compiler configuration files 19
Creating custom configuration files 20

Chapter 3. Compiler options reference 23
Summary of compiler options by functional category 23
Output control 23
Input control 24
Language element control 25
Template control (C++ only) 26
Floating-point and integer control 26
Object code control 27
Error checking and debugging 28
Listings, messages, and compiler information . . 31
Optimization and tuning 31
Linking 33
Portability and migration 34
Compiler customization 34
Individual option descriptions 35
-### (-#) (pound sign) 36
-+ (plus sign) (C++ only) 37
--help (-qhelp) 37
--version (-qversion) 38
@*file* (-qoptfile) 40
-B 42
-C, -C! 43
-D 44
-E 45
-F. 46
-I 48
-L 49
-O, -qoptimize 50
-P 52
-R 53
-S. 54
-U 55
-X (-W) 56
-Werror (-qhalt) 57
-Wunsupported-xl-macro 58
-c 59
-dM (-qshowmacros) 60
-e 60
-fasm (-qasm) 61
-fcommon (-qcommon) 63
-fdollars-in-identifiers (-qdollar) 64
-fdump-class-hierarchy (-qdump_class_hierarchy) (C++ only) 65
-finline-functions (-qinline) 65
-fPIC , -fpic (-qpic) 69
-fpack-struct (-qalign) 70
-fsigned-bitfields, -funsigned-bitfields (-qbitfields) 71
-fsigned-char, -funsigned-char (-qchars) 71
-fstrict-aliasing (-qalias=ansi), -qalias 72
-fsyntax-only (-qsyntaxonly) 74
-ftemplate-depth (-qtemplatedepth) (C++ only) 75
-ftls-model (-qtls) 76
-ftime-report (-qphsinfo) 77
-ftree-vectorize (-qsimd) 78
-g. 80
-gdwarf (-qdbgfmt) 83
-include (-qinclude) 84
-isystem (-qc_stdinc) (C only) 85
-isystem (-qcpp_stdinc) (C++ only) 87
-isystem (-qgcc_c_stdinc) (C only) 88
-isystem (-qgcc_cpp_stdinc) (C++ only) 89
-l 91
-march (-qarch) 92
-mtune (-qtune) 94

© Copyright IBM Corp. 2015

-mzvector . . . 96
-m31, -m64 (-q31, -q64) . . . 96
-o. . . . 97
-p, -pg, -qprofile . . . 98
-qasm_as . . . 99
-qcrt, -nostartfiles (-qnocrt) . . . 100
-qeh (C++ only) . . . 101
-qfloat . . . 102
-qfullpath . . . 103
-qfuncsect . . . 104
-qhot . . . 104
-qinitauto. . . . 106
-qipa . . . 109
-qisolated_call . . . 114
-qkeepparm . . . 115
-qlib, -nodefaultlibs (-qnolib) . . . 116
-qlibansi . . . 117
-qlinedebug . . . 118
-qlist . . . 119
-qmakedep, -MD (-qmakedep=gcc) . . . 120
-qpath . . . 122
-qpdf1, -qpdf2 . . . 123
-qpriority (C++ only) . . . 129
-qreport . . . 130
-qrtti, -fno-rtti (-qnortti) (C++ only) . . . 131
-qsaveopt. . . . 132
-qshowpdf . . . 134
-qsmallstack . . . 135
-qstaticinline (C++ only) . . . 136
-qstdinc, -qnostdinc (-nostdinc, -nostdinc++) . . 137
-qtimestamps . . . 138
-qtmplinst (C++ only) . . . 139
-r . . . 139
-s . . . 140
-shared (-qmkshrobj) . . . 141
-static (-qstaticlink) . . . 142
-std (-qlanglvl) . . . 144
-t . . . 148
-v, -V . . . 150
-w . . . 150
-x (-qsourcetype) . . . 151
Supported GCC options . . . 153

Chapter 4. Compiler pragmas reference . . . 159

Pragma directive syntax . . . 159
Scope of pragma directives . . . 159
Supported GCC pragmas . . . 160
Supported IBM pragmas . . . 160
#pragma disjoint . . . 161
#pragma execution_frequency . . . 162
#pragma nosimd . . . 163
#pragma nounroll . . . 164
#pragma option_override . . . 164
#pragma pack . . . 166
#pragma reachable . . . 169

Chapter 5. Compiler predefined macros . . . 171

General macros. . . . 171
Macros indicating the XL C/C++ compiler . . . 172
Macros related to the platform . . . 173
Macros related to compiler features . . . 174
Macros related to compiler option settings. . . 174
Macros related to architecture settings . . . 175
Macros related to language levels . . . 176
Unsupported macros from other XL compilers . . 177

Chapter 6. Compiler built-in functions 179

Fixed-point built-in functions . . . 179
Absolute value functions . . . 179
Population count functions . . . 179
Cache-related built-in functions . . . 180
Data cache functions . . . 180
Block-related built-in functions . . . 181
bzero . . . 181
Vector built-in functions . . . 181
GCC atomic memory access built-in functions (IBM extension) . . . 181
Atomic lock, release, and synchronize functions 182
Atomic fetch and operation functions . . . 183
Atomic operation and fetch functions . . . 186
Atomic compare and swap functions . . . 189
Miscellaneous built-in functions . . . 190
Optimization-related functions . . . 190
Memory-related functions . . . 190
Hardware built-in functions . . . 191
__cp . . . 191
__cvb . . . 192
__cvbg . . . 193
__cvd . . . 193
__cvdg . . . 193
__dp . . . 194
__fidbr . . . 195
__fiebr. . . . 196
__fixbr . . . 197
__lcbb . . . 198
__mp . . . 198
__srp . . . 199
__stck . . . 200
__stckf. . . . 201
__zap . . . 201
Transactional memory built-in functions . . . 203
Transaction begin and end functions. . . . 204
Transaction abort functions . . . 205
Transaction inquiry functions . . . 206
Transaction store functions . . . 211

Notices . . . 213

Trademarks . . . 215

Index . . . 217

About this document

This document is a reference for the IBM® XL C/C++ for Linux on z Systems™, V1.2 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, built-in functions, environment variables, error messages, and return codes.

Who should read this document

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

How to use this document

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

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

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

- Compiler installation: see the *XL C/C++ Installation Guide*.
- The C or C++ programming language: see the *XL C/C++ Language Reference* for information about the syntax, semantics, and IBM implementation of the C or C++ IBM extension features. See C/C++ standards for the details of standard features.
- Programming topics: see the *XL C/C++ Optimization and Programming Guide* for detailed information about developing applications with XL C/C++, with a focus on program portability and optimization.

How this document is organized

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

© Copyright IBM Corp. 2015

Chapter 2, "Configuring compiler defaults," on page 17 discusses topics related to setting up default compilation settings, including setting environment variables and customizing the configuration file.

Chapter 3, "Compiler options reference," on page 23 provides a summary of options according to their functional category, through which you can look up and link to options by function. This chapter also includes individual descriptions of selected compiler option sorted alphabetically and a list of the rest of supported GCC options.

Chapter 4, "Compiler pragmas reference," on page 159 provides a list of GCC supported pragmas, which are sorted alphabetically. Then it provides the detailed information of each IBM supported pragma.

Chapter 5, "Compiler predefined macros," on page 171 provides a list of compiler macros grouped according to their category. It also provides a list of compiler macros that might be supported by other XL compilers but are not supported in IBM XL C/C++ for Linux on z Systems, V1.2.

Chapter 6, "Compiler built-in functions," on page 179 contains individual descriptions of XL C/C++ built-in functions for Power® architectures, categorized by their functionality.

Conventions

Typographical conventions

The following table shows the typographical conventions used in the IBM XL C/C++ for Linux on z Systems, V1.2 information.

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Lowercase commands, executable names, compiler options, and directives.	The compiler provides basic invocation commands, **xlc** and **xlC** (**xlc++**), along with several other compiler invocation commands to support various C/C++ language levels and compilation environments.
italics	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	Make sure that you update the *size* parameter if you return more than the *size* requested.
underlining	The default setting of a parameter of a compiler option or directive.	nomaf | maf
`monospace`	Programming keywords and library functions, compiler builtins, examples of program code, command strings, or user-defined names.	To compile and optimize myprogram.c, enter: `xlc myprogram.c -O3`.

Qualifying elements (icons)

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

Table 2. Qualifying elements

Qualifier/Icon	Meaning
C only begins ▶ C C ◀ C only ends	The text describes a feature that is supported in the C language only; or describes behavior that is specific to the C language.
C++ only begins ▶ C++ C++ ◀ C++ only ends	The text describes a feature that is supported in the C++ language only; or describes behavior that is specific to the C++ language.
IBM extension begins ▶ IBM IBM ◀ IBM extension ends	The text describes a feature that is an IBM extension to the standard language specifications.
C11 begins ▶ C11 C11 ◀ C11 ends	The text describes a feature that is introduced into standard C as part of C11.
C++11 begins ▶ C++11 C++11 ◀ C++11 ends	The text describes a feature that is introduced into standard C++ as part of C++11.
C++14 begins ▶ C++14 C++14 ◀ C++14 ends	The text describes a feature that is introduced into standard C++ as part of C++14.

Syntax diagrams

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

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

  The ▶▶—— symbol indicates the beginning of a command, directive, or statement.

  The ——▶ symbol indicates that the command, directive, or statement syntax is continued on the next line.

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

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

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

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

```
►►──keyword──required_argument───────────────────────────────────────────►◄
```

- Optional items are shown below the main path:

```
►►──keyword──┬───────────────────┬───────────────────────────────────────►◄
             └─optional_argument─┘
```

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

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

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

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

```
►►──keyword──┬────────────────────┬──────────────────────────────────────►◄
             ├─optional_argument1─┤
             └─optional_argument2─┘
```

- An arrow returning to the left above the main line (a repeat arrow) indicates that you can make more than one choice from the stacked items or repeat an item. The separator character, if it is other than a blank, is also indicated:

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

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

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

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

Example of a syntax statement

EXAMPLE *char_constant* {*a*|*b*}[*c*|*d*]*e*[,*e*]... *name_list*{*name_list*}...

The following list explains the syntax statement:

- Enter the keyword EXAMPLE.

- Enter a value for *char_constant*.
- Enter a value for *a* or *b*, but not for both.
- Optionally, enter a value for *c* or *d*.
- Enter at least one value for *e*. If you enter more than one value, you must put a comma between each.
- Optionally, enter the value of at least one *name* for *name_list*. If you enter more than one value, you must put a comma between each *name*.

Note: The same example is used in both the syntax-statement and syntax-diagram representations.

Examples in this information

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

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

Related information

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

IBM XL C/C++ information

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

- Quick Start Guide

  The Quick Start Guide (`quickstart.pdf`) is intended to get you started with IBM XL C/C++ for Linux on z Systems, V1.2. It is located by default in the XL C/C++ directory and in the `\quickstart` directory of the installation DVD.
- README files

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

  Man pages are provided for the compiler invocations and all command-line utilities provided with the product. Instructions for installing and accessing the man pages are provided in the *IBM XL C/C++ for Linux on z Systems, V1.2 Installation Guide*.
- Online product documentation

  The fully searchable HTML-based documentation is viewable in IBM Knowledge Center at http://www.ibm.com/support/knowledgecenter/SSVUN6_1.2.0/com.ibm.compilers.loz.doc/welcome.html.
- PDF documents

  PDF documents are available on the web at http://www.ibm.com/support/docview.wss?uid=swg27044043.

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

Table 3. XL C/C++ PDF files

Document title	PDF file name	Description
IBM XL C/C++ for Linux on z Systems, V1.2 Installation Guide, GC27-5995-01	`install.pdf`	Contains information for installing XL C/C++ and configuring your environment for basic compilation and program execution.
Getting Started with IBM XL C/C++ for Linux on z Systems, V1.2, GI13-2865-01	`getstart.pdf`	Contains an introduction to the XL C/C++ product, with information about setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
IBM XL C/C++ for Linux on z Systems, V1.2 Compiler Reference, SC27-5998-01	`compiler.pdf`	Contains information about the various compiler options, pragmas, macros, environment variables, and built-in functions.
IBM XL C/C++ for Linux on z Systems, V1.2 Language Reference, SC27-5996-01	`langref.pdf`	Contains information about language extensions for portability and conformance to nonproprietary standards.
IBM XL C/C++ for Linux on z Systems, V1.2 Optimization and Programming Guide, SC27-5997-01	`proguide.pdf`	Contains information about advanced programming topics, such as application porting, library development, application optimization, and the XL C/C++ high-performance libraries.

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

More information related to XL C/C++, including IBM Redbooks® publications, white papers, and other articles, is available on the web at http://www.ibm.com/support/docview.wss?uid=swg27044043.

For more information about C/C++, see the C/C++ café at https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=5894415f-be62-4bc0-81c5-3956e82276f3.

Standards and specifications

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

- *Information Technology - Programming languages - C, ISO/IEC 9899:1990*, also known as *C89*.
- *Information Technology - Programming languages - C, ISO/IEC 9899:1999*, also known as *C99*.
- *Information Technology - Programming languages - C, ISO/IEC 9899:2011*, also known as *C11*.
- *Information Technology - Programming languages - C++, ISO/IEC 14882:1998*, also known as *C++98*.
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2003*, also known as *C++03*.
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2011*, also known as *C++11* (Partial support).

- *Information Technology - Programming languages - C++, ISO/IEC 14882:2014*, also known as *C++14* (Partial support).
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985*.

Other information

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

Technical support

Additional technical support is available from the XL C/C++ Support page at http://www.ibm.com/support/entry/portal/product/rational/xl_c/c++_for_linux_on_z_systems. This page provides a portal with search capabilities to a large selection of Technotes and other support information.

If you cannot find what you need, you can send an email to compinfo@ca.ibm.com.

For the latest information about XL C/C++, visit the product information site at http://www.ibm.com/software/products/en/xlcpp-loz.

How to send your comments

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

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

Chapter 1. Compiling and linking applications

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

- Preprocessing of program source
- Compiling and assembling into object files
- Linking into an executable

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

The following sections describe how to invoke the XL C/C++ compiler to preprocess, compile, and link source files and libraries:

- "Invoking the compiler"
- "Types of input files" on page 3
- "Types of output files" on page 4
- "Specifying compiler options" on page 5
- "Preprocessing" on page 8
- "Linking" on page 10
- "Compiler messages and listings" on page 12

Invoking the compiler

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

All the invocation commands allow for threadsafe compilations. You can use them to link the programs that use multithreading.

Note: For each invocation command, the compiler configuration file defines default option settings and, in some cases, macros; for information about the defaults implied by a particular invocation, see the /opt/ibm/xlC/1.2/etc/xlc.cfg.*$OSRelease.gcc$gccVersion* file for your system. For example, /opt/ibm/xlC/1.2/etc/xlc.cfg.sles.12.gcc.4.8.2

Table 4. Compiler invocations

Invocations	Description	Equivalent invocations
xlc	Invokes the compiler for C source files. This command supports all of the ISO C99 standard features, and most IBM language extensions. This invocation is recommended for all applications.	xlc_r
c99	Invokes the compiler for C source files. This command supports all ISO C99 language features, but does not support IBM language extensions. Use this invocation for strict conformance to the C99 standard.	c99_r

© Copyright IBM Corp. 2015

Table 4. Compiler invocations (continued)

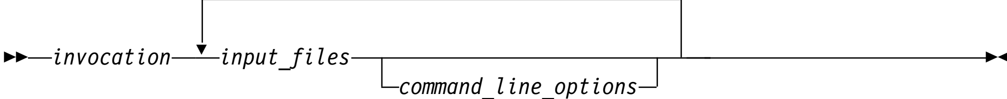
Invocations	Description	Equivalent invocations
c89	Invokes the compiler for C source files. This command supports all ANSI C89 language features, but does not support IBM language extensions. Use this invocation for strict conformance to the C89 standard.	c89_r
cc	Invokes the compiler for C source files. This command supports pre-ANSI C, and many common language extensions. You can use this command to compile legacy code that does not conform to standard C.	cc_r
xlc++, xlC	Invokes the compiler for C++ source files. If any of your source files are C++, you must use this invocation to link with the correct runtime libraries. Files with .c suffixes, assuming you have not used the **-+** compiler option, are compiled as C language source code.	xlc++_r, xlC_r

Related information

- "-std (-qlanglvl)" on page 144

Command-line syntax

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



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

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

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

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

Related information

- "Types of input files"

Types of input files

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

By default, the compiler preprocesses and compiles all the specified source files. Although you usually want to use this default, you can use the compiler to preprocess the source file without compiling; see "Preprocessing" on page 8 for details.

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

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

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

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

Preprocessed source files
: Preprocessed files are useful for checking macros and preprocessor directives. Preprocessed C source files have a `.i` suffix and preprocessed C++ source files have a `.ii` suffix, for example, *file_name*`.i` and *file_name*`.ii`. The compiler sends the preprocessed source file, *file_name*`.i` or *file_name*`.ii`, 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, library files, and unstripped executable files serve as input to the linker. After compilation, the linker links all of the specified object files to create an executable file.

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

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

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

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

Related information:

"Input control" on page 24

Types of output files

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

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

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

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

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

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

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

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

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

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

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

Related information:

"Output control" on page 23

Specifying compiler options

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

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

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

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

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

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

Option conflicts that do not follow this priority sequence are described in "Resolving conflicting compiler options" on page 6.

Specifying compiler options on the command line

Most options specified on the command line override both the default settings of the option and options set in the configuration file. Similarly, most options specified on the command line are in turn overridden by pragma directives, which provide you a means of setting compiler options right in the source file. Options that do not follow this scheme are listed in "Resolving conflicting compiler options" on page 6.

Specifying compiler options in a configuration file

The default configuration file (/opt/ibm/xlC/1.2/etc/ xlc.cfg.*$OSRelease.gcc$gccVersion*, for example, /opt/ibm/xlC/1.2/etc/ xlc.cfg.sles.12.gcc.4.8.3) 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 19.

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 35, 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 159.

Other pragmas do not have equivalent command-line options; these are described in detail throughout Chapter 4, "Compiler pragmas reference," on page 159.

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*, **-L***directory*, and **-R***directory_path* options.

When options such as **-qfloat** 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 statements in source code override compiler options specified on the command line.
2. Compiler options specified on the command line override compiler options specified as environment variables or in a configuration file. If conflicting or incompatible compiler options are specified on the command line, the option appearing later on the command line takes precedence.

3. Compiler options specified as environment variables override compiler options specified in a configuration file.
4. Compiler options specified in a configuration file override compiler default settings.

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

Option	Conflicting options	Resolution
-E	-P, -S	-E
-P	-c, -o, -S	-P
-#	-v	-#
-F	-B, -t, -W, -qpath	-B, -t, -W, -qpath
-qpath	-B, -t	-qpath
-S	-c	-S
-nostdinc, -nostdinc++ (-qnostdinc)	-isystem (-qc_stdinc, -qcpp_stdinc, -qgcc_c_stdinc, -qgcc_cpp_stdinc)	-nostdinc, -nostdinc++ (-qnostdinc)

Specifying compiler options for architecture-specific compilation

You can use the **-m31**, **-m64**, **-march**, and **-mtune** compiler options to optimize the output of the compiler to suit:
- The broadest possible selection of target processors
- A range of processors within a given processor architecture family
- A single specific processor

Generally speaking, the options do the following:
- **-m31** selects 31-bit execution mode.
- **-m64** selects 64-bit execution mode.
- **-march** selects the general family processor architecture for which instruction code should be generated.
- **-mtune** selects the specific processor for which compiler output is optimized. The **-mtune** option influences only the performance of the code when running on a particular system but does not determine where the code will run.

The compiler evaluates compiler options in the following order, with the last allowable one found determining the compiler mode:
1. Internal default (64-bit mode)
2. Configuration file settings
3. Command line compiler options (**-m31**, **-m64**, **-march**, and **-mtune**)

The compilation mode actually used by the compiler depends on a combination of the settings of the **-m31**, **-m64**, **-march**, and **-mtune** compiler options, subject to the following conditions:
- *Compiler mode* is set according to the last-found instance of the **-m31** or **-m64** compiler options.

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

Allowable combinations of these options are found in "-mtune (-qtune)" on page 94.

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

- **-march** option is incompatible with user-selected **-mtune** option.

  Resolution: Compiler issues a warning message, and sets **-mtune** to the **-march** setting's default **-mtune** value.
- Selected **-march** or **-mtune** options are not known to the compiler.

  Resolution: Compiler issues a warning message, sets **-march** and **-mtune** to their default settings. The compiler mode (31-bit or 64-bit) is determined by the **-m31** or **-m64** compiler settings.

Related information

- "-march (-qarch)" on page 92
- "-mtune (-qtune)" on page 94
- "-m31, -m64 (-q31, -q64)" on page 96

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 45	Preprocesses the source files and writes the output to standard output. By default, `#line` directives are generated.
"-P" on page 52	Preprocesses the source files and creates an intermediary file with a .i file name suffix for each source file. By default, `#line` directives are not generated.

Option	Description
"-C, -C!" on page 43	Preserves comments in preprocessed output.
"-D" on page 44	Defines a macro name from the command line, as if in a `#define` directive.
-dD[1]	Emits macro definitions to preprocessed output and prints the output.
"-dM (-qshowmacros)" on page 60[1]	Emits macro definitions to preprocessed output.
"-qmakedep, -MD (-qmakedep=gcc)" on page 120	Produces the dependency files that are used by the **make** tool for each source file.
-M[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 .d file with the name of the input file.
-MF *file*[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 .d file with the name of the input file. 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 *target*[1]	Changes the target of the rule emitted by dependency generation and quotes any characters that are special to the **make** tool.
-MT *target*[1]	Changes the target of the rule emitted by dependency generation.
"-U" on page 55	Undefines a macro name defined by the compiler or by the **-D** option.

Note:

1. For details about the option, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

Directory search sequence for included files

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

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

- User-defined header files

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

- Use the standard `#include <file_name>` preprocessor directive in the including source file.
- Use the standard `#include "file_name"` preprocessor directive in the including source file.
- Use the **-include** compiler option.

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

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

1. Header files included with **-include** only: The compiler searches the current (working) directory from which the compiler is invoked.
2. Header files included with **-include** 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, but the search path can be changed with the **-isystem** (**-qgcc_c_stdinc** or **-qgcc_cpp_stdinc**) option.[1]

Note:

1. If the **-nostdinc** or **-nostdinc++** (**-qnostdinc**) compiler option is in effect, step 4 is omitted.

Related information

- "-I" on page 48
- "-isystem (-qc_stdinc) (C only)" on page 85
- "-isystem (-qcpp_stdinc) (C++ only)" on page 87
- "-isystem (-qgcc_c_stdinc) (C only)" on page 88
- "-isystem (-qgcc_cpp_stdinc) (C++ only)" on page 89
- "-include (-qinclude)" on page 84
- "-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 137

Linking

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

- **-c**
- **-E**
- **-M**
- **-P**
- **-S**

- **-fsyntax-only (-qsyntaxonly)**
- **-### (-#)**
- **--help (-qhelp)**
- **--version (-qversion)**

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

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

```
xlc myfile.c -o myfile
```

If you use the **-shared (-qmkshrobj)** option to create a shared library, the default name of the shared object created is `a.out`. You can use the **-o** option to rename the file and give it a `.so` suffix.

If you use XL C/C++ to compile and link the assembler files that are produced by GCC, specify the **-march** option to target the same architecture level as the one used by GCC. Note that the default architecture levels of XL C/C++ and GCC are different.

You can invoke the linker explicitly with the **ld** command. However, the compiler invocation commands set several linker options, and link some standard files into the executable output by default. In most cases, it is better to use one of the compiler invocation commands to link your object files. For a complete list of options available for linking, see "Linking" on page 33.

Note: If you want to use a nondefault linker, you can use either of the following approaches:

- Use **-t** and **-B** or use **-qpath** to specify the nondefault linker, for example,

  ```
  -tl -Blinker_path
  ```

  or

  ```
  -qpath=l:linker_path
  ```

- Customize the configuration file of the compiler to use the nondefault linker. For more information about how to customize the configuration file, see Using custom compiler configuration files and Creating custom configuration files.

Related information

- "-march (-qarch)" on page 92
- "-shared (-qmkshrobj)" on page 141

Order of linking

The compiler links libraries in the following order:

1. System startup libraries

2. User .o files and libraries
3. XL C/C++ libraries
4. C++ standard libraries
5. C standard libraries

Related information

- "Linking" on page 33
- "Redistributable libraries"

Redistributable libraries

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

- Ship the packages that contain the redistributable libraries with your application. The packages are stored under the `images/rpms` directory in the installed compiler package..
- Direct the users of your application to download the appropriate runtime libraries from the *Latest updates for supported IBM C and C++ compilers* link from the XL C/C++ support website at http://www.ibm.com/support/entry/portal/product/rational/xl_c/c++_for_linux_on_z_systems.

For information about the licensing requirements related to the distribution of these packages, see the LicenseAgreement.pdf, LicenseInformation.pdf, and redist.txt files in the installed compiler package.

Table 5. Redistributable libraries

Package name	Libraries (and default installation path)	Description

libxlc-devel	`/opt/ibm/xlC/1.2.0/lib/libibmc++.a`
	`/opt/ibm/xlC/1.2.0/lib/libxl.a`
	`/opt/ibm/xlC/1.2.0/lib/libxlopt.a`
	`/opt/ibm/xlC/1.2.0/lib64/libibmc++.a`
	`/opt/ibm/xlC/1.2.0/lib64/libxl.a`
	`/opt/ibm/xlC/1.2.0/lib64/libxlopt.a`
		XL C/C++ compiler libraries

libxlc	`/opt/ibm/lib/libibmc++.so.1` `/opt/ibm/lib64/libibmc++.so.1`	XL C++ runtime libraries

libatlas-devel	`/opt/ibm/atlas/1.2.0/include`
	`/opt/ibm/atlas/1.2.0/lib`
	`/opt/ibm/atlas/1.2.0/lib64`
		Automatically Tuned Linear Algebra Software (ATLAS) libraries

Compiler messages and listings

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

- "Compiler messages" on page 13
- "Compiler listings" on page 13
- "Paging space errors during compilation" on page 15

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++ 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 **-Werror (-qhalt=w)** option to stop the compilation for warnings and all types of errors.

You can use the **-Werror=unused-command-line-argument** option to switch between warnings and errors for invalid options.

Table 6. 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.
C 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.
S	Severe error	error	Compilation continues, but object code is not generated. The compiler cannot correct the error conditions that are found. • If the message indicates a resource limit (for example, file system full or paging space full), provide additional resources and recompile. • If the message indicates that different compiler options are needed, recompile using 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.
C U	Unrecoverable error	fatal error	The compiler halts. An internal compile-time error has occurred. Report the message to your IBM service representative.

Related information

- “-Werror (-qhalt)” on page 57
- “Listings, messages, and compiler information” on page 31

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 any of the following options, which provide different types of information:

- -qlist
- -qreport

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

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

File table section
: Lists the file name and number for each main source file and include file. Each file is associated with a file number, starting with the main source file, which is assigned file number 0.

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

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

  Relevance of profiling data
  : This section shows the relevance of the profiling data to the source code during the **-qpdf1** phase. The relevance is indicated by a number in the range of 0 - 100. The larger the number is, the more relevant the profiling data is to the source code, and the more performance gain can be achieved by using the profiling data.

  Missing profiling data
  : This section might include a warning message about missing profiling data. The warning message is issued for each function for which the compiler does not find profiling data.

  Outdated profiling data
  : This section might include a warning message about outdated profiling data. The compiler issues this warning message for each function that is modified after the **-qpdf1** phase. The warning message is also issued when the optimization level changes from the **-qpdf1** phase to the **-qpdf2** phase.

Data reorganization section
: Displays data reorganization messages for program variable data during the IPA link pass when **-qreport** is used with **-qipa=level=2**. Reorganization information includes:

  - array splitting
  - array transposing
  - memory allocation merging
  - array interleaving
  - array coalescing

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

Constant area section
If you specify the **-qlist** option, the Constant area section lists the constants used in the program. The compiler loads from the constant area section by loading the starting address of this section and adding the fixed offsets to the respective constants.

Related information

- "Listings, messages, and compiler information" on page 31

Paging space errors during compilation

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

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

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

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

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

Chapter 2. Configuring compiler defaults

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

- Internally defined settings. These settings are predefined by the compiler and you cannot change them.
- Settings defined by system environment variables. Certain environment variables are required by the compiler; others are optional. You might have already set some of the basic environment variables during the installation process. For more information, see the XL C/C++ Installation Guide. "Setting environment variables" provides a complete list of the required and optional environment variables you can set or reset after installing the compiler.
- Settings defined in the compiler configuration file, `xlc.cfg`. The compiler requires many settings that are determined by its configuration file. Normally, the configuration file is automatically generated during the installation procedure. For more information, see the XL C/C++ Installation Guide. However, you can customize this file after installation, to specify additional compiler options, default option settings, library search paths, and other settings. Information on customizing the configuration file is provided in "Using custom compiler configuration files" on page 19.

Setting environment variables

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

```
variable=value
export variable
```

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

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

```
setenv variable value
```

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

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

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

- "Compile-time and link-time environment variables" on page 18
- "Runtime environment variables" on page 18

© Copyright IBM Corp. 2015

Compile-time and link-time environment variables

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

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

LD_RUN_PATH
: Specifies search paths for dynamically loaded libraries, equivalent to using the **-R** link-time option. The shared-library locations named by the environment variable are embedded into the executable, so the dynamic linker can locate the libraries at application run time. For more information about this environment variable, see your operating system documentation. See also "-R" on page 53.

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

PATH
: Specifies the directory search path for the executable files of the compiler. Executables are in /opt/ibm/xlC/1.2.0/bin/ if installed to the default location. For information, see Setting the PATH environment variable to include the path to the XL C/C++ invocations in the *XL C/C++ Installation Guide*

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

XLC_USR_CONFIG
: Specifies the location of a custom configuration file to be used by the compiler. The file name must be given with its absolute path. The compiler will first process the definitions in this file before processing those in the default system configuration file, or those in a customized file specified by the **-F** option; for more information, see "Using custom compiler configuration files" on page 19.

Runtime environment variables

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

LD_LIBRARY_PATH
: Specifies an alternate directory search path for dynamically linked libraries at application run time. If shared libraries required by your application have been moved to an alternate directory that was not specified at link

time, and you do not want to relink the executable, you can set this environment variable to allow the dynamic linker to locate them at run time. For more information about this environment variable, see your operating system documentation.

PDFDIR
Optionally specifies the directory in which profiling information is saved when you run an application that you have compiled with the **-qpdf1** option. The default value is unset, and the compiler places the profile data file in the current working directory. If the PDFDIR environment variable is set but the specified directory does not exist, the compiler issues a warning message. When you recompile or relink your program with the **-qpdf2** option, the compiler uses the data saved in this directory to optimize the application. It is recommended that you set this variable to an absolute path if you use profile-directed feedback (PDF). See "-qpdf1, -qpdf2" on page 123 for more information.

PDF_WL_ID

This environment variable is used to distinguish the sets of PDF counters that are generated by multiple training runs of the user program. Each run receives distinct input.

By default, PDF counters for training runs after the first training run are added to the first and the only set of PDF counters. This behavior can be changed by setting the PDF_WL_ID environment variable before each PDF training run. You can set PDF_WL_ID to an integer value in the range 1 - 65535. The PDF runtime library then uses this number to tag the set of PDF counters that are generated by this training run. After all the training runs complete, the PDF profile file contains multiple sets of PDF counters, each set with an ID number.

Using custom compiler configuration files

The XL C/C++ compiler generates a default configuration file /opt/ibm/xlC/1.2/etc/xlc.cfg.*$OSRelease.gcc$gccVersion* at installation time (for example, /opt/ibm/xlC/1.2/etc/xlc.cfg.sles.12.gcc.4.8.2). (See the *XL C/C++ Installation Guide* for more information on the various tools you can use to generate the configuration file during installation.) The configuration file specifies information that the compiler uses when you invoke it.

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

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

You have several options for customizing configuration files:

- You can directly edit the default configuration file. In this case, the customized options will apply for all users for all compilations. The disadvantage of this option is that you will need to reapply your customizations to the new default configuration file that is provided every time you install a compiler update.

- You can use the default configuration file as the basis of customized copies that you specify at compile time with the **-F** option. In this case, the custom file overrides the default file on a per-compilation basis.

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

Related reference:

"-F" on page 46

Related information:

"Compile-time and link-time environment variables" on page 18

Creating custom configuration files

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

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

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

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

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

Figure 1. Sample configuration file

In this example:

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

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

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

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

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

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

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

DEFLT:
    options=<C>
```

Figure 3. Default configuration file xlc.cfg

Overriding the default order of attribute values

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

Table 7. Assignment operators and attribute ordering

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

Table 7. Assignment operators and attribute ordering (continued)

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

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

Custom user-defined configuration file ~/userconfig2

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

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

Default configuration file xlc.cfg

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

DEFLT:
    options=<C>
```

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

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

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

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

Figure 4. Using additional assignment operations

Examples of stanzas in custom configuration files

Stanza	Description
`DEFLT: use=DEFLT` `options = -g`	This example specifies that the **-g** option is to be used in all compilations.
`xlc: use=xlc    options+=-qlist`	This example specifies that **-qlist** is to be used for any compilation called by the **xlc** command. This **-qlist** specification overrides the default setting of **-qlist** specified in the system configuration file.
`DEFLT: use=DEFLT` `libraries=-L/home/user/lib,-lmylib`	This example specifies that all compilations should link with /home/user/lib/libmylib.a.

Chapter 3. Compiler options reference

This section contains a summary of the compiler options available in XL C/C++ by functional category, followed by detailed descriptions of the individual options.

Related information

• "Specifying compiler options" on page 5

Summary of compiler options by functional category

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

- "Output control"
- "Input control" on page 24
- "Language element control" on page 25
- "Template control (C++ only)" on page 26
- "Floating-point and integer control" on page 26
- "Error checking and debugging" on page 28
- "Listings, messages, and compiler information" on page 31
- "Optimization and tuning" on page 31
- "Object code control" on page 27
- "Linking" on page 33
- "Portability and migration" on page 34
- "Compiler customization" on page 34

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 8. Compiler output options

Option name	Description
"-c" on page 59	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, -C!" on page 43	When used in conjunction with the **-E** or **-P** options, preserves or removes comments in preprocessed output.
"-dM (-qshowmacros)" on page 60	Emits macro definitions to preprocessed output.
"-E" on page 45	Preprocesses the source files named in the compiler invocation, without compiling.

© Copyright IBM Corp. 2015

Table 8. Compiler output options (continued)

Option name	Description
"-o" on page 97	Specifies a name for the output object, assembler, executable, or preprocessed file.
"-P" on page 52	Preprocesses the source files named in the compiler invocation, without compiling, and creates an output preprocessed file for each input file.
"-qmakedep, -MD (-qmakedep=gcc)" on page 120	Produces the dependency files that are used by the **make** tool for each source file.
"-qtimestamps" on page 138	Controls whether or not implicit time stamps are inserted into an object file.
"-shared (-qmkshrobj)" on page 141	Creates a shared object from generated object files.
"-S" on page 54	Generates an assembler language file for each source file.
"-X (-W)" on page 56	**-Xpreprocessor** *option* or **-Wp,***option* passes the listed option directly to the preprocessor.

The following options are supported by XL C/C++ for GCC compatibility. For details about these options, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

- -###
- -d*CHARS*
- -M
- -MD
- -MF *file*
- -MG
- -MM
- -MMD
- -MP
- -MQ *target*
- -MT *target*
- -Xpreprocessor

Input control

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

Table 9. Compiler input options

Option name	Description
"-include (-qinclude)" on page 84	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.
"-I" on page 48	Adds a directory to the search path for include files.

Table 9. Compiler input options (continued)

Option name	Description
"-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 137	Specifies whether the standard include directories are included in the search paths for system and user header files.
"-x (-qsourcetype)" on page 151	Instructs the compiler to treat all recognized source files as a specified source type, regardless of the actual file name suffix.

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 10. Language element control options

Option name	Description
"-D" on page 44	Defines a macro as in a `#define` preprocessor directive.
"-fasm (-qasm)" on page 61	Controls the interpretation and subsequent generation of code for assembler language extensions.
"-fdollars-in-identifiers (-qdollar)" on page 64	Allows the dollar-sign ($) symbol to be used in the names of identifiers.
"-mzvector" on page 96	Enables the compiler support for vector programming including the `vector` and `__vector` keywords and vector built-in functions.
"-qstaticinline (C++ only)" on page 136	Controls whether inline functions are treated as having `static` or `extern` linkage.
"-std (-qlanglvl)" on page 144	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 55	Undefines a macro defined by the compiler or by the **-D** compiler option.
"-X (-W)" on page 56	**-Xassembler** *option* or **-Wa,***option* passes the listed option directly to the assembler.

The following options are supported by XL C/C++ for GCC compatibility. For details about these options, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

- -ansi
- -fconstexpr-depth
- -fconstexpr-steps
- -fexec-charset
- -ffreestanding
- -fgnu89-inline
- -fhosted
- -fno-access-control

- -fno-builtin
- -fno-gnu-keywords
- -fno-operator-names
- -fno-rtti
- -fpermissive
- -fsigned-bitfields
- -fsigned-char
- -ftemplate-backtrace-limit
- -ftemplate-depth
- -funsigned-bitfields
- -funsigned-char
- -trigraphs
- -Xassembler

Template control (C++ only)

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

Table 11. C++ template options

Option name	Description
"-ftemplate-depth (-qtemplatedepth) (C++ only)" on page 75	Specifies the maximum number of recursively instantiated template specializations that will be processed by the compiler.
"-qtmplinst (C++ only)" on page 139	Manages the implicit instantiation of templates.

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 12. Floating-point and integer control options

Option name	Description
"-fsigned-bitfields, -funsigned-bitfields (-qbitfields)" on page 71	Specifies whether bit fields are signed or unsigned.
"-fsigned-char, -funsigned-char (-qchars)" on page 71	Determines whether all variables of type `char` is treated as `signed` or `unsigned`.
"-qfloat" on page 102	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 13. Object code control options

Option name	Description
"-fcommon (-qcommon)" on page 63	Controls where uninitialized global variables are allocated.
"-fPIC , -fpic (-qpic)" on page 69	Generates position-independent code suitable for use in shared libraries.
"-ftls-model (-qtls)" on page 76	Enables recognition of the `__thread` storage class specifier, which designates variables that are to be allocated thread-local storage; and specifies the threadlocal storage model to be used.
"-m31, -m64 (-q31, -q64)" on page 96	Selects either 31-bit or 64-bit compiler mode.
"-qeh (C++ only)" on page 101	Controls whether exception handling is enabled in the module being compiled.
"-qfuncsect" on page 104	Places instructions for each function in a separate section. Placing each function in its own section might reduce the size of your program because the linker can collect garbage per function rather than per object file.
"-qpriority (C++ only)" on page 129	Specifies the priority level for the initialization of static objects.
"-qrtti, -fno-rtti (-qnortti) (C++ only)" on page 131	Generates runtime type identification (RTTI) information for exception handling and for use by the `typeid` and `dynamic_cast` operators.
"-qsaveopt" on page 132	Saves the command-line options used for compiling a source file, the user's configuration file name and the options specified in the configuration files, the version and level of each compiler component invoked during compilation, and other information to the corresponding object file.
"-r" on page 139	Produces a nonexecutable output file to use as an input file in another ld command call. This file may also contain unresolved symbols.
"-s" on page 140	Strips the symbol table, line number information, and relocation information from the output file.

The following options are supported by XL C/C++ for GCC compatibility. For details about these options, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

- -fpack-struct
- -fpic, -fno-pic

- -fpie, -fno-pie
- -fPIE, -fno-PIE
- -fshort-wchar
- -mtpf-trace

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 31.

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

Table 14. Error checking and debugging options

Option name	Description
"-### (-#) (pound sign)" on page 36	Previews the compilation steps specified on the command line, without actually invoking any compiler components.
"-g" on page 80	Generates debugging information for use by a symbolic debugger, and makes the program state available to the debugging session at selected source locations.
"-gdwarf (-qdbgfmt)" on page 83	Specifies the format for the debugging information in object files.
"-qfullpath" on page 103	When used with the **-g** or **-qlinedebug** option, this option records the full, or absolute, path names of source and include files in object files compiled with debugging information, so that debugging tools can correctly locate the source files.
"-qinitauto" on page 106	Initializes uninitialized automatic variables to a specific value, for debugging purposes.
"-qkeepparm" on page 115	When used with **-O2** or higher optimization, specifies whether procedure parameters are stored on the stack.
"-qlinedebug" on page 118	Generates only line number and source file name information for a debugger.
"-Werror (-qhalt)" on page 57	Stops compilation before producing any object, executable, or assembler source files if the maximum severity of compile-time messages equals or exceeds the severity you specify.
"-Wunsupported-xl-macro" on page 58	Checks whether any unsupported XL macro is used.

Options to control diagnostic messages formatting

The following options are supported by XL C/C++ for GCC compatibility. For details about these options, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

- -fansi-escape-codes
- -fcolor-diagnostics
- -fdiagnostics-format=[clang|msvc|vi]
- -fdiagnostics-fixit-info
- -fdiagnostics-print-source-range-info
- -fdiagnostic-parsable-fixits
- -fdiagnostic-show-category=[none|id|name]
- -fdiagnostics-show-name
- -fdiagnostic-show-template-tree
- -fmessage-length
- -fno-diagnostics-show-caret
- -fno-diagnostics-show-option
- -fno-elide-type
- -fshow-column
- -fshow-source-location
- -pedantic
- -pedantic-errors
- -Wambiguous-member-template
- -Wbind-to-temporary-copy
- -Wextra-tokens

Options to request or suppress warnings

The following options are supported by XL C/C++ for GCC compatibility. For details about these options, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

- -fsyntax-only
- -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
- -Wpedantic -pedantic -pedantic-errors
- -Wpadded
- -Wparantheses
- -Wpointer-arith
- -Wpointer-sign
- -Wreorder
- -Wreturn-type
- -Wsequence-point
- -Wshadow

- -Wsign-compare
- -Wsign-conversion
- -Wsizeof-pointer-memaccess
- -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 28 to provide a more robust overview of your application when checking for errors and unexpected behavior.

Table 15. Listings and messages options

Option name	Description
"-fdump-class-hierarchy (-qdump_class_hierarchy) (C++ only)" on page 65	Dumps a representation of the hierarchy and virtual function table layout of each class object to a file.
"-qlist" on page 119	Produces a compiler listing file that includes object and constant area sections.
"-qreport" on page 130	Produces listing files that show how sections of code have been optimized.
"--help (-qhelp)" on page 37	Displays the man page of the compiler.
"--version (-qversion)" on page 38	Displays the version and release of the compiler being invoked.

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.

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

Table 16. Optimization and tuning options

Option name	Description
"-finline-functions (-qinline)" on page 65	Attempts to inline functions instead of generating calls to those functions, for improved performance.
"-ftree-vectorize (-qsimd)" on page 78	Controls whether the compiler can automatically take advantage of vector instructions for processors that support them. **Equivalent pragma:** #pragma nosimd
"-fstrict-aliasing (-qalias=ansi), -qalias" on page 72	Indicates whether a program contains certain categories of aliasing or does not conform to C/C++ standard aliasing rules. The compiler limits the scope of some optimizations when there is a possibility that different names are aliases for the same storage location.
"-march (-qarch)" on page 92	Specifies the processor architecture for which the code (instructions) should be generated.
-mtune (-qtune)	Tunes instruction selection, scheduling, and other architecture-dependent performance enhancements to run best on a specific hardware architecture.
"-O, -qoptimize" on page 50	Specifies whether to optimize code during compilation and, if so, at which level.
"-qhot" on page 104	Performs high-order loop analysis and transformations (HOT) during optimization.
"-qipa" on page 109	Enables or customizes a class of optimizations known as interprocedural analysis (IPA).
"-qisolated_call" on page 114	Specifies functions in the source file that have no side effects other than those implied by their parameters.
"-qlibansi" on page 117	Assumes that all functions with the name of an ANSI C library function are in fact the system functions.
"-qpdf1, -qpdf2" on page 123	Tunes optimizations through *profile-directed feedback* (PDF), where results from sample program execution are used to improve optimization near conditional branches and in frequently executed code sections.
"-qshowpdf" on page 134	When used with **-qpdf1** and a minimum optimization level of **-O2** at compile and link steps, creates a PDF map file that contains additional profiling information for all procedures in your application.

Table 16. Optimization and tuning options (continued)

Option name	Description
"-qsmallstack" on page 135	Minimizes stack usage where possible. Disables optimizations that increase the size of the stack frame.

The following options are supported by XL C/C++ for GCC compatibility. For details about these options, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

- --sysroot
- -isysroot
- -isystem

Linking

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

Table 17. Linking options

Option name	Description
"-e" on page 60	When used together with the **-shared (-qmkshrobj)** option , specifies an entry point for a shared object.
"-L" on page 49	At link time, searches the directory path for library files specified by the **-l** option.
"-l" on page 91	Searches for the specified library file. The linker searches for *libkey.so*, and then *libkey.a* if *libkey.so* is not found.
"-qcrt, -nostartfiles (-qnocrt)" on page 100	Specifies whether system startup files are to be linked.
"-qlib, -nodefaultlibs (-qnolib)" on page 116	Specifies whether standard system libraries and XL C/C++ libraries are to be linked.
"-R" on page 53	At link time, writes search paths for shared libraries into the executable, so that these directories are searched at program run time for any required shared libraries.
"-static (-qstaticlink)" on page 142	Controls whether static or shared runtime libraries are linked into an application.
"-X (-W)" on page 56	**-Xlinker** *option* or **-Wl,***option* passes the listed option directly to the linker.

The following options are supported by XL C/C++ for GCC compatibility. For details about these options, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

- -idirafter
- -imacros
- -iprefix

- -iquote
- -iwithprefix
- -pie
- -rdynamic
- -Xlinker

Portability and migration

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

Table 18. Portability and migration option

Option name	Description
"-fpack-struct (-qalign)" on page 70	Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data.

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 19. Compiler customization options

Option name	Description
"@*file* (-qoptfile)" on page 40	Specifies a file containing a list of additional command line options to be used for the compilation.
"-B" on page 42	Specifies substitute path names for XL C/C++ components such as the assembler, C preprocessor, and linker.
"-F" on page 46	Names an alternative configuration file or stanza for the compiler.
"-isystem (-qc_stdinc) (C only)" on page 85	Changes the standard search location for the XL C header files.
"-isystem (-qcpp_stdinc) (C++ only)" on page 87	Changes the standard search location for the XL C++ header files.
"-isystem (-qgcc_c_stdinc) (C only)" on page 88	Changes the standard search location for the GNU C system header files.
"-isystem (-qgcc_cpp_stdinc) (C++ only)" on page 89	Changes the standard search location for the GNU C++ system header files.
"-qasm_as" on page 99	Specifies the path and flags used to invoke the assembler in order to handle assembler code in an `asm` assembly statement.
"-qpath" on page 122	Specifies substitute path names for XL C/C++ components such as the compiler, assembler, linker, and preprocessor.

Table 19. Compiler customization options (continued)

Option name	Description
"-t" on page 148	Applies the prefix specified by the **-B** option to the designated components.
"-X (-W)" on page 56	Passes the listed options to a component that is executed during compilation.

Individual option descriptions

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

For each option, the following information is provided:

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

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

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 159

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 171

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

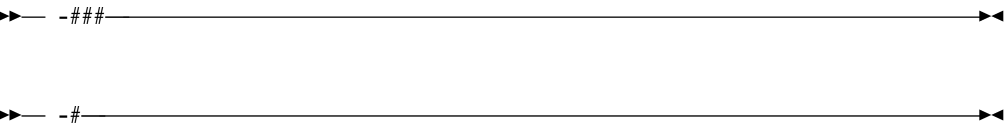
None.

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 linker that would be invoked, and the default options that would be specified for each program. The preprocessor, compiler, and linker are not invoked.

Syntax



Usage

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

This option displays the same information as **-v**, but 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:

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

Related information

- "-v, -V" on page 150

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

Category

Input control

Pragma equivalent

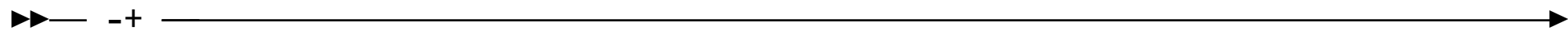
None.

Purpose

Compiles any file as a C++ language file.

This option is equivalent to the **-x c++** option.

Syntax



Usage

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

You cannot use the **-+** option with the **-qsourcetype** or **-x** option.

Predefined macros

None.

Examples

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

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

Related information

- "-x (-qsourcetype)" on page 151

--help (-qhelp)

Category

Listings, messages, and compiler information

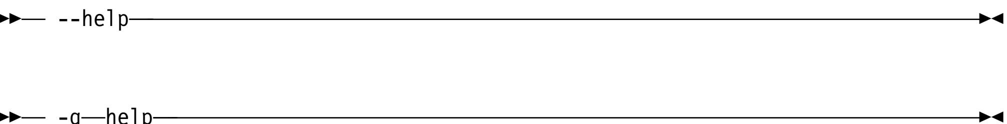
Pragma equivalent

None.

Purpose

Displays the man page of the compiler.

Syntax



Usage

If you specify the **--help (-qhelp)** option, regardless of whether you provide input files, the compiler man page is displayed and the compilation stops.

Predefined macros

None.

Related information

- "--version (-qversion)"

--version (-qversion)

Category

Listings, messages, and compiler information

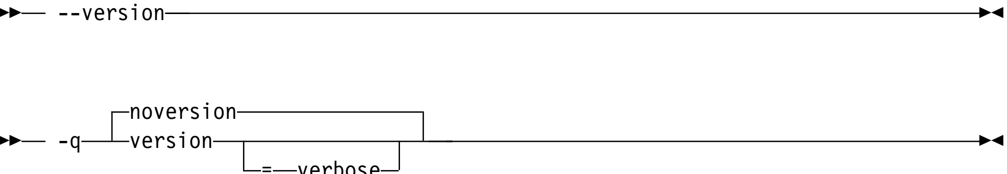
Pragma equivalent

None.

Purpose

Displays the version and release of the compiler being invoked.

Syntax



Defaults

-qnoversion

--version is not set by default.

Parameters

verbose
: Displays information about the version, release, and level of each compiler component installed.

Usage

When you specify **--version (-qversion)**, the compiler displays the version information and exits; compilation is stopped. If you want to save this information to the output object file, you can do so with the **-qsaveopt -c** options.

-qversion specified without the **verbose** suboption shows compiler information in the format:

```
product_nameVersion: VV.RR.MMMM.LLLL
```

where:

V Represents the version.
R Represents the release.
M Represents the modification.
L Represents the level.

For more details, see Example 1.

-qversion=verbose shows component information in the following format:

```
component_name Version: VV.RR(product_name) Level: component_build_date ID:
component_level_ID
```

where:

component_name
: Specifies an installed component, such as the low-level optimizer.

component_build_date
: Represents the build date of the installed component.

component_level_ID
: Represents the ID associated with the level of the installed component.

For more details, see Example 2.

Predefined macros

None.

Example 1

The output of specifying the **--version (-qversion)** option:

```
IBM XL C/C++ for Linux on z Systems, V1.1
Version: 01.01.0000.0000
```

Example 2

The output of specifying the **-qversion=verbose** option:

```
IBM XL C/C++ for Linux on z Systems, V1.2
Version: 01.02.0000.0000
Driver Version: 01.02(C/C++): 141017 ID:  _kUiP60sMEeSe5NK2G0fWzA
C/C++ Front End Version: 01.02(C/C++) Level: 141002 ID:  _INkPkEnXEeSe5NK2G0fWzA
High-Level Optimizer Version: 01.02(C/C++)Level: 141003 ID: __Iiw0kr_EeSe5NK2G0fWzA
Low-Level Optimizer Version: 01.02(C/C++)Level: 141003 ID: _zoirQEqlEeSe5NK2G0fWzA
```

Related information

- "-qsaveopt" on page 132

@*file* (-qoptfile)

Category

Compiler customization

Pragma equivalent

None.

Purpose

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

Syntax

Defaults

None.

Parameters

filename
: Specifies the name of the file 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 option file follows these rules:

- Specify the options you want to include in the file with the same syntax as on the command line. The option 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 @*file* **(-qoptfile)** option from the command line, and sequentially inserts the options included in the file before the other subsequent options that you specify.

The *@file* **(-qoptfile)** option is also valid within an option file. The files that contain another option file are processed in a depth-first manner. The compiler avoids infinite loops by detecting and ignoring cycles in option file inclusion.

If *@file* **(-qoptfile)** and **-qsaveopt** are specified on the same command line, the original command line is used for **-qsaveopt**. A new line for each option file is included representing the contents of each option file. The options contained in the file are saved to the compiled object file.

Predefined macros

None.

Example 1

This is an example of specifying an option file.

```
$ cat options.file
# To perform optimization at -O3 level, and high-order
# loop analysis and transformations during optimization
-O3 -qhot
# To generate position-independent code
-fPIC

$ xlC -qlist @options.file -qipa test.c
```

The preceding example is equivalent to the following invocation:

```
$ xlC -qlist -O3 -qhot -fPIC -qipa test.c
```

Example 2

This is an example of specifying an option file that contains *@file* **(-qoptfile)** with a cycle.

```
$ cat options.file2
# To perform optimization at -O3 level, and high-order
# loop analysis and transformations during optimization
-O3 -qhot
# To include the -qoptfile option in the same option file
@options.file2
# To generate position-independent code
-fPIC
# To produce a compiler listing file
-qlist

$ xlC -qlist @options.file2 -qipa test.c
```

The preceding example is equivalent to the following invocation:

```
$ xlC -qlist -O3 -qhot -fPIC -qlist -qipa test.c
```

Example 3

This is an example of specifying an option file that contains *@file* **(-qoptfile)** without a cycle.

```
$ cat options.file1
-O3 -qhot
@options.file2
-qalias=ansi
```

```
$ cat options.file2
-qchars=signed

$ xlC @options.file1 test.c
```

The preceding example is equivalent to the following invocation:

```
$ xlC -O3 -qhot -qchars=signed test.c
```

Example 4

This is an example of specifying **-qsaveopt** and *@file* **(-qoptfile)** on the same command line.

```
$ cat options.file3
-O3
-qhot

$ xlC -qsaveopt -qipa @options.file3 test.c -c

$ what test.o
test.o:
opt f xlC -qsaveopt -qipa @options.file3 test.c -c
optfile options.file3 -O3 -qhot
```

Related information

- "-qsaveopt" on page 132

-B

Category

Compiler customization

Pragma equivalent

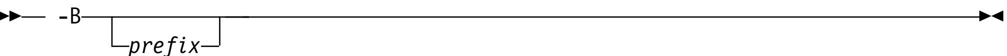
None.

Purpose

Specifies substitute path names for XL C/C++ components such as the assembler, C preprocessor, and linker.

You can use this option if you want to keep multiple levels of some or all of the XL C/C++ executables and have the option of specifying which one you want to use. However, it is preferred that you use the **-qpath** option to accomplish this instead.

Syntax



Defaults

The default paths for the compiler executables are defined in the compiler configuration file.

Parameters

prefix
Defines part of a path name for programs you can name with the **-t** option. You must add a slash (/). If you specify the **-B** option without the *prefix*, the default prefix is /lib/o.

Usage

The **-t** option specifies the programs to which the **-B** prefix name is to be appended; see "-t" on page 148 for a list of these. If you use the **-B** option without **-t***programs*, the prefix you specify applies to all of the compiler executables.

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

Predefined macros

None.

Examples

In this example, an earlier level of the compiler components is installed in the default installation directory. To test the upgraded product before making it available to everyone, the system administrator restores the latest installation image under the directory `/home/jim` and then tries it out with commands similar to:

```
xlc -tcbI -B/home/jim/opt/ibm/xlC/1.2.0/bin/ test_suite.c
```

Once the upgrade meets the acceptance criteria, the system administrator installs it in the default installation directory.

Related information

- "-qpath" on page 122
- "-t" on page 148
- "Invoking the compiler" on page 1
- The **-B** option that GCC provides. For details, see the GCC online documentation at http://gcc.gnu.org/onlinedocs/.

-C, -C!

Category

Output control

Pragma equivalent

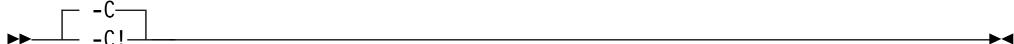
None.

Purpose

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

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

Syntax



Defaults

-C

Usage

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

You can use the **-C!** option to override the **-C** option specified in a default makefile or configuration file.

Predefined macros

None.

Examples

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

```
xlc myprogram.c -P -C
```

Related information

- "-E" on page 45
- "-P" on page 52

-D

Category

Language element control

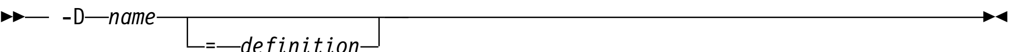
Pragma equivalent

None.

Purpose

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

Syntax



Defaults

Not applicable.

Parameters

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

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

Usage

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

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

Predefined macros

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

Examples

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

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

Related information

- "-U" on page 55
- Chapter 5, "Compiler predefined macros," on page 171

-E

Category

Output control

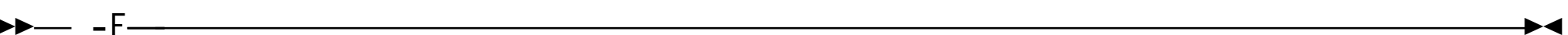
Pragma equivalent

None.

Purpose

Preprocesses the source files named in the compiler invocation, without compiling.

Syntax



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:

```
xlc myprogram.c -E
```

If `myprogram.c` has a code fragment such as:

```
#define SUM(x,y) (x + y)
int a ;
#define mm 1   /* This is a comment in a
                  preprocessor directive */
int b ;        /* This is another comment across
                  two lines */
int c ;
               /* Another comment */
c = SUM(a,b) ;  /* Comment in a macro function argument*/
```

the output will be:

```
int a ;

int b ;

int c ;

c = a + b ;
```

Related information

- "-C, -C!" on page 43
- "-P" on page 52
- "-fsyntax-only (-qsyntaxonly)" on page 74

-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. For example, if you are compiling with **xlc**, but you specify the **c99** stanza, the compiler will use all the settings specified in the **c99** stanza.

Usage

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

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

Predefined macros

None.

Examples

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

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

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

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

To compile `myprogram.c` using the stanza `c99` you have created in a configuration file called `/usr/tmp/myconfig.cfg`, enter:

```
xlc myprogram.c -F/usr/tmp/myconfig.cfg:c99
```

Related information

- "Using custom compiler configuration files" on page 19
- "-B" on page 42
- "-t" on page 148
- "-X (-W)" on page 56
- "Specifying compiler options in a configuration file" on page 5
- "Compile-time and link-time environment variables" on page 18

-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 9 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 **-nostdinc** or **-nostdinc++** (**-qnostdinc**) is in effect, the compiler searches *only* the paths specified by the **-I** option for header files, and not the standard search paths as well.

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

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

Predefined macros

None.

Examples

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

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

Related information

- "-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 137
- "-include (-qinclude)" on page 84
- "Directory search sequence for included files" on page 9
- "Specifying compiler options in a configuration file" on page 5

-L

Category

Linking

Pragma equivalent

None.

Purpose

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

Syntax

```
►►— -L—directory_path————————————————————————————————————►◄
```

Defaults

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

Parameters

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

Usage

Paths specified with the **-L** compiler option are only searched at link time. To specify paths that should be searched at run time, use the **-R** option.

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

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**.

For more information, refer to the **ld** documentation for your operating system.

Predefined macros

None.

Examples

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

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

Related information

- "-l" on page 91
- "-R" on page 53

-O, -qoptimize

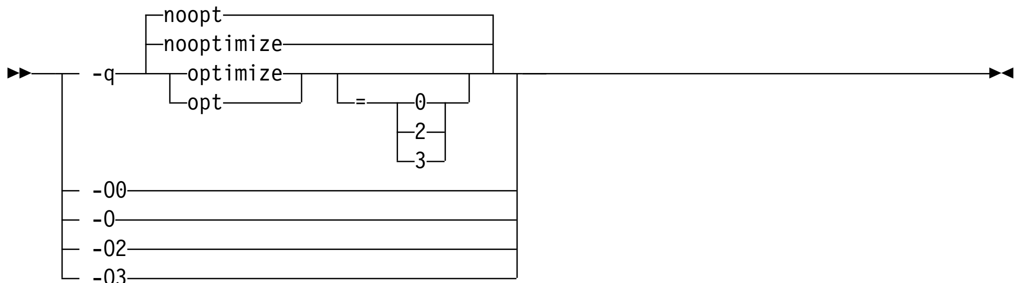
Category

Optimization and tuning

Purpose

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

Syntax



Defaults

-qnooptimize or **-O0** or **-qoptimize=0**

Parameters

`-O0 | nooptimize | noopt | optimize|opt=0`
: Performs only quick local optimizations such as constant folding and elimination of local common subexpressions.

`-O | -O2 | optimize | opt | optimize|opt=2`
: Performs optimizations that the compiler developers considered the best combination for compilation speed and runtime performance. The optimizations may change from product release to release. If you need a specific level of optimization, specify the appropriate numeric value.

`-O3 | optimize|opt=3`
: Performs additional optimizations that are memory intensive, compile-time

intensive, or both. They are recommended when the desire for runtime improvement outweighs the concern for minimizing compilation resources.

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

1. 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.
2. Specifying **-O3** implies **-qhot=level=0**, unless you explicitly specify **-qhot** or **-qhot=level=1** option.

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

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

When **-O3** and **-qhot=level=1** are in effect, the compiler replaces any calls in the source code to standard math library functions with calls to the equivalent MASS library functions.

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.

If optimization level **-O3** is specified on the command line, the **-qhot** and **-qipa** options that are set by the optimization level cannot be overridden by `#pragma option_override(`*identifier*`, "opt(level, 0)")` or `#pragma option_override(`*identifier*`, "opt(level, 2)")`.

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** is in effect. Otherwise, it is undefined.

Examples

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

```
xlc myprogram.c -O3
```

Related information

- “-qhot” on page 104
- “-qipa” on page 109

- "-qpdf1, -qpdf2" on page 123
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*.
- "#pragma option_override" on page 164

-P

Category

Output control

Pragma equivalent

None.

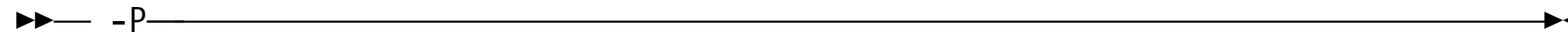
Purpose

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.

Note: This option is not equivalent to the GCC option **-P** .

Syntax



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, -C!" on page 43
- "-E" on page 45
- "-fsyntax-only (-qsyntaxonly)" on page 74

-R

Category

Linking

Pragma equivalent

None.

Purpose

At link time, writes search paths for shared libraries into the executable, so that these directories are searched at program run time for any required shared libraries.

Syntax

```
►►── -R──directory_path──────────────────────────────────────────►◄
```

Defaults

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

Usage

If the **-R***directory_path* option is specified both in the configuration file and on the command line, the paths specified in the configuration file are searched first at run time.

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

Predefined macros

None.

Examples

To compile `myprogram.c` so that the directory `/usr/tmp/old` is searched at run time along with standard directories for the dynamic library `libspfiles.so`, enter:

```
xlc myprogram.c -lspfiles -R/usr/tmp/old
```

Related information

- "-L" on page 49

-S

Category

Output control

Pragma equivalent

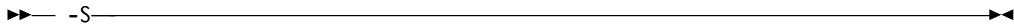
None.

Purpose

Generates an assembler language file for each source file.

The resulting file has a .s suffix and can be assembled to produce object .o files or an executable file (a.out).

Syntax



Defaults

Not applicable.

Usage

You can invoke the assembler with any compiler invocation command. For example,

```
xlc myprogram.s
```

will invoke the assembler, and if successful, the linker to create an executable file, a.out.

If you specify **-S** with **-E** or **-P**, **-E** or **-P** takes precedence. Order of precedence holds regardless of the order in which they were specified on the command line.

You can use the **-o** option to specify the name of the file produced only if no more than one source file is supplied. For example, the following is *not* valid:

```
xlc myprogram1.c myprogram2.c -o -S
```

Predefined macros

None.

Examples

To compile `myprogram.c` to produce an assembler language file `myprogram.s`, enter:

```
xlc myprogram.c -S
```

To assemble this program to produce an object file `myprogram.o`, enter:

```
xlc myprogram.s -c
```

To compile `myprogram.c` to produce an assembler language file `asmprogram.s`, enter:

```
xlc myprogram.c -S -o asmprogram.s
```

Related information

- "-E" on page 45
- "-P" on page 52

-U

Category

Language element control

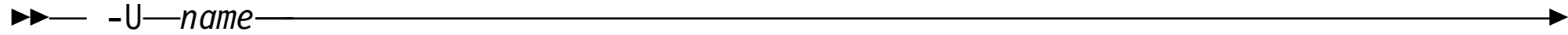
Pragma equivalent

None.

Purpose

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

Syntax



Defaults

Many macros are predefined by the compiler; see Chapter 5, "Compiler predefined macros," on page 171 for those that can be undefined (that is, are not *protected*). The compiler configuration file also uses the **-D** option to predefine several macro names for specific invocation commands; for details, see the configuration file for your system.

Parameters

name
: The macro you want to undefine.

Usage

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

The **-U***name* option has a higher precedence than the **-D***name* option.

Predefined macros

None.

Examples

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

```
xlc myprogram.c  -U__unix
```

Related information

- "-D" on page 44

-X (-W)

Category

Compiler customization

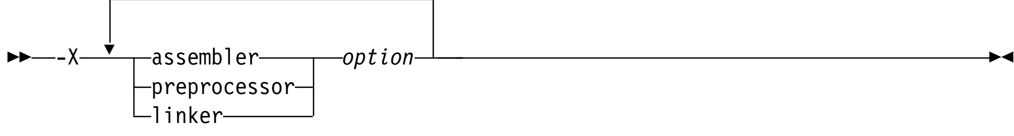
Pragma equivalent

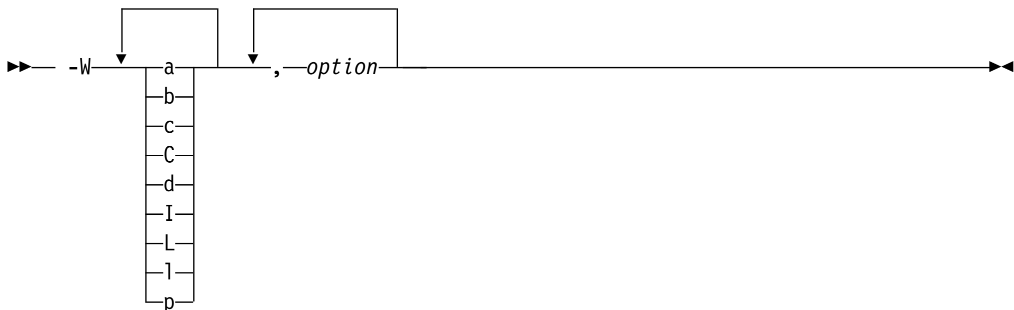
None.

Purpose

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

Syntax





Parameters

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

Note: For **-X**, for details about the options for linking and assembling, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/

The following table shows the correspondence between **-X** or **-W** parameters and the component names:

Parameter of -W	Parameter of -X	Description	Component name
a	assembler	The assembler	as
b		The low-level optimizer	xlCcode
c, C		The C and C++ compiler front end	xlCentry

Parameter of -W	Parameter of -X	Description	Component name
d		The disassembler	dis
I (uppercase i)		The high-level optimizer, compile step	ipa
L		The high-level optimizer, link step	ipa
l (lowercase L)	linker	The linker	ld
p	preprocessor	The preprocessor	xlCentry

Usage

In the string following the **-W** option, use a comma as the separator for each option, and do not include any spaces. For the **-X** option, one space is needed before the *option*. 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 **-X** or **-W** option in the configuration file, you can use the escape sequence backslash comma (\,) to represent a comma in the parameter string.

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

Predefined macros

None.

Examples

To compile the file `file.c` and pass the linker option **-symbolic** to the linker, enter the following command:

```
xlc -Xlinker -symbolic file.c
```

To compile the file `uses_many_symbols.c` and the assembly file `produces_warnings.s` so that `produces_warnings.s` is assembled with the assembler option **-alh**, and the object files are linked with the option **-s** (write list of object files and strip final executable file), issue the following command:

```
xlc -Xassembler -alh produces_warnings.s -Xlinker -s uses_many_symbols.c
```

Related information

- "Invoking the compiler" on page 1

-Werror (-qhalt)

Category

Error checking and debugging

Purpose

Stops compilation before producing any object, executable, or assembler source files if the maximum severity of compile-time messages equals or exceeds the severity you specify.

Syntax

Defaults

By default, **-Werror (-qhalt=w)** is disabled.

Parameters

w Specifies that compilation is to stop for warnings (W) and all types of errors.

Predefined macros

None.

Examples

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

```
xlc myprogram.c -Werror
```

-Wunsupported-xl-macro

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Checks whether any unsupported XL macro is used.

Syntax

```
▶▶──-Wunsupported-xl-macro──────────────────────────────────────────▶◀
```

Defaults

By default, **-Wunsupported-xl-macro** is disabled.

Usage

Some macros that might be supported by other XL compilers are unsupported in IBM XL C/C++ for Linux on z Systems, V1.2.

You can specify the **-Wunsupported-xl-macro** option to check whether any unsupported macro is used. If an unsupported macro is used, the compiler issues a warning message.

Predefined macros

None.

Related information

"Unsupported macros from other XL compilers" on page 177

-c

Category

Output control

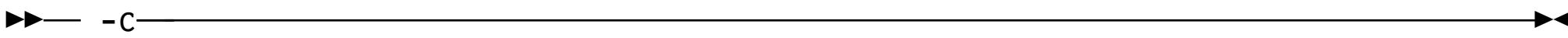
Pragma equivalent

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



Defaults

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

Usage

When this option is in effect, the compiler creates an output object file, *file_name*.o, for each valid source file, such as *file_name*.c, *file_name*.i, *file_name*.C, *file_name*.cpp, 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:

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

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

```
xlc myprogram.c -c -o new.o
```

Related information

- "-E" on page 45
- "-o" on page 97
- "-P" on page 52
- "-fsyntax-only (-qsyntaxonly)" on page 74

-dM (-qshowmacros)

Category

"Output control" on page 23

Pragma equivalent

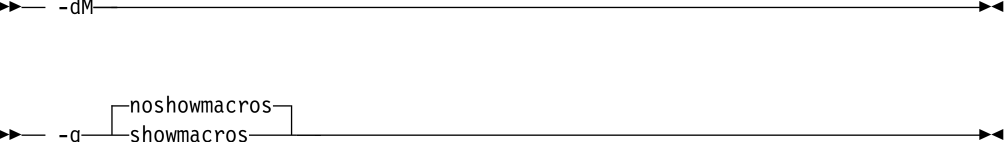
None

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.
- Only macros defined internally by the preprocessor are considered predefined; all other macros are considered as user-defined.

Related information

- "-E" on page 45
- "-P" on page 52

-e

Category

Linking

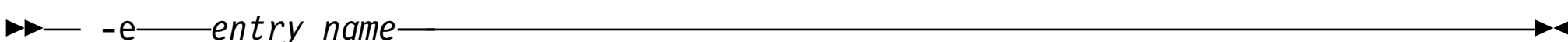
Pragma equivalent

None.

Purpose

Specifies an entry point for a shared object when used together with the **-shared (-qmkshrobj)** option.

Syntax



Defaults

None.

Parameters

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

Usage

Specify the **-e** option only with the **-shared (-qmkshrobj)** option.

Note: When you link object files, do not use the **-e** option. The default entry point of the executable output is `__start`. Changing this label with the **-e** flag can produce errors.

Predefined macros

None.

Related information

- "-shared (-qmkshrobj)" on page 141

-fasm (-qasm)

Category

Language element control

Pragma equivalent

None.

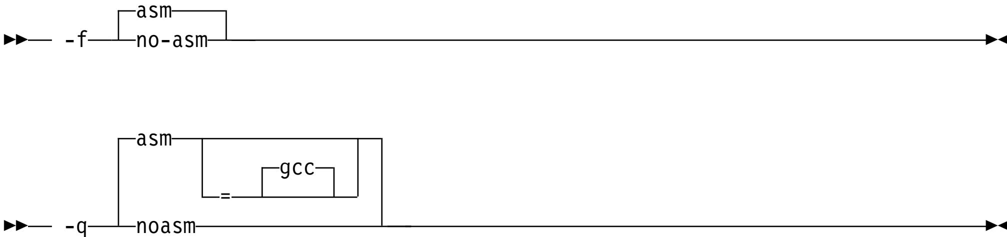
Purpose

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

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

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

Syntax



Defaults

-qasm=gcc or -fasm

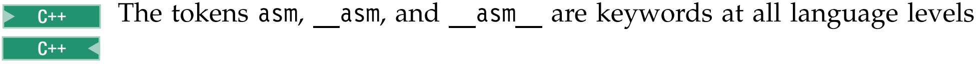
Parameters

gcc
: Instructs the compiler to recognize the extended GCC syntax and semantics for assembly statements.

Specifying **-qasm** without a suboption is equivalent to specifying the default.

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



The tokens `asm`, `__asm`, and `__asm__` are keywords at all language levels.

For detailed information about the syntax and semantics of inline `asm` statements, see "Inline assembly statements" in the *XL C/C++ Language Reference*.

Examples

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

```
int a,b;
int main() {
   a = 11;
   b = 44;
   asm("AR %0, %1" : "+r"(a) : "r"(b));
return a;
}
```

Related information

- “-qasm_as” on page 99
- “-std (-qlanglvl)” on page 144
- "Inline assembly statements" in the *XL C/C++ Language Reference*

-fcommon (-qcommon)

Category

Object code control

Pragma equivalent

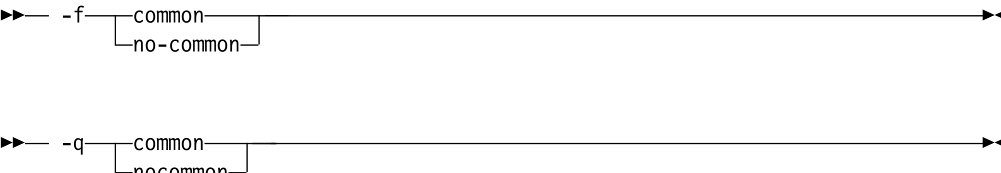
None.

Purpose

Controls where uninitialized global variables are allocated.

When **-fcommon (-qcommon)** is in effect, uninitialized global variables are allocated in the common section of the object file. When **-fno-common (-qnocommon)** is in effect, uninitialized global variables are initialized to zero and allocated in the data section of the object file.

Syntax



Defaults

- C **-fcommon (-qcommon)** except when **-shared (-qmkshrobj)** is specified; **-fno-common (-qnocommon)** when **-shared (-qmkshrobj)** is specified.
- C++ **-fno-common (-qnocommon)**

Usage

This option does not affect static or automatic variables, or the declaration of structure or union members.

This option is overridden by the `common|nocommon` and `section` variable attributes. See "The common and nocommon variable attribute" and "The section variable attribute" in the *XL C/C++ Language Reference*.

Predefined macros

None.

Examples

In the following declaration, where `a` and `b` are global variables:

```
int a, b;
```

Compiling with **-fcommon (-qcommon)** produces the equivalent of the following assembly code:

```
.common a,4,4
.common b,4,4
```

Compiling with **-fno-common (-qnocommon)** produces the equivalent of the following assembly code:

```
.globl a
.size a,4
a:
.fill 4

.globl b
.size b,4
b:
.fill 4
```

Related information

- "-shared (-qmkshrobj)" on page 141
- "The common and nocommon variable attribute" in the *XL C/C++ Language Reference*
- "The section variable attribute" in the *XL C/C++ Language Reference*

-fdollars-in-identifiers (-qdollar)

Category

Language element control

Pragma equivalent

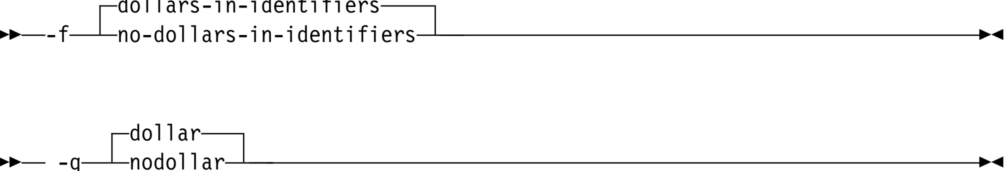
None

Purpose

Allows the dollar-sign ($) symbol to be used in the names of identifiers.

When **-fdollars-in-identifiers** or **-qdollar** is in effect, the dollar symbol $ in an identifier is treated as a base character.

Syntax



Defaults

-fdollars-in-identifiers or **-qdollar**

Predefined macros

None.

Examples

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

```
xlc myprogram.c -fdollars-in-identifiers
```

Related information

- "-std (-qlanglvl)" on page 144

-fdump-class-hierarchy (-qdump_class_hierarchy) (C++ only)

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Dumps a representation of the hierarchy and virtual function table layout of each class object to a file.

Syntax

Defaults

Not applicable.

Usage

The output file name consists of the source file name appended with a .class suffix.

Predefined macros

None.

Examples

To compile `myprogram.C` to produce a file named `myprogram.C.class` containing the class hierarchy information, enter:

```
xlc++ myprogram.C -fdump-class-hierarchy
```

-finline-functions (-qinline)

Category

Optimization and tuning

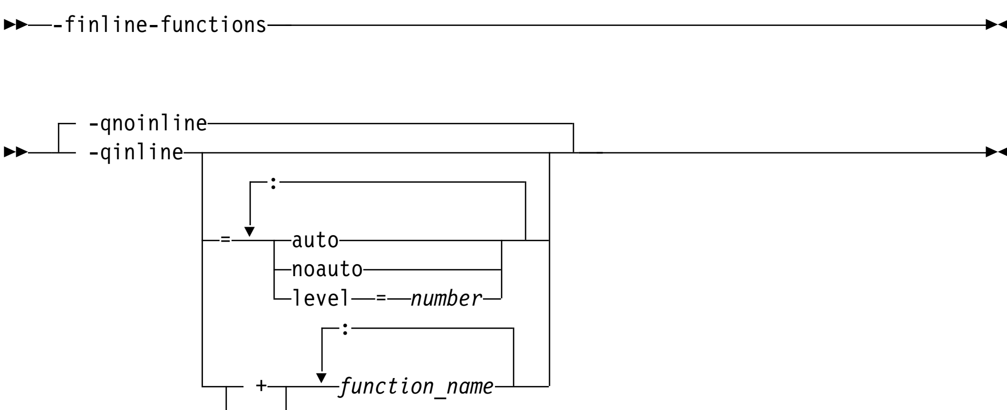
Pragma equivalent

None.

Purpose

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

Syntax



Defaults

If **-qinline** is not specified, the default option is **-qnoinline** at the **-O0** or **-qnoopt** optimization level, or **-qinline=noauto:level=5** at the **-O2** or higher optimization level.

If **-qinline** is specified without any suboptions, the default option is **-qinline=auto:level=5**.

Parameters

auto | noauto
: Enables or disables automatic inlining. When option **-qinline=auto** is in effect, all functions are considered for inlining by the compiler. When option **-qinline=noauto** is in effect, only the following types of functions are considered for inlining:

- Functions that are defined with the inline specifier
- Small functions that are identified by the compiler

The compiler determines whether a function is appropriate for inlining, and enabling automatic inlining does not guarantee that a function is inlined.

level=*number*
: Indicates the relative degree of inlining. The values for *number* must be integers in the range 0 - 10 inclusive. The default value for *number* is 5. The greater the value of *number*, the more aggressive inlining the compiler conducts.

function_name
: If *function_name* is specified after the **-qinline+** option, the named function must be inlined. If *function_name* is specified after the **-qinline-** option, the named function must not be inlined. C++ The *function_name* must be the mangled name of the function. You can find the mangled function name in the listing file. C++

Usage

You can specify C++ **-qinline** C++ or specify **-qinline** with any optimization level of C++ **-O** C++ , **-O2**, or **-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.

You can use the **-qinline+<***function_name***>** or **-qinline-<***function_name***>** option to specify the functions that must be inlined or must not be inlined.

IBM The **-qinline-<***function_name***>** option takes higher precedence than the `always_inline` or `__always_inline__` attribute. When you specify both the `always_inline` or `__always_inline__` attribute and the **-qinline-<***function_name***>** option to a function, that function is not inlined. IBM

Specifying **-qnoinline** disables all inlining, including that achieved by the high-level optimizer with the **-qipa** option, and 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
- Functions that are specified with the **-qinline+<***function_name***>** option

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

Predefined macros

None.

Examples

Example 1

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

```
xlc myprogram.c -O2 -qnoinline
```

However, if some functions in `myprogram.c` are specified with IBM the `always_inline` or `__always_inline__` attribute IBM , the **-qnoinline** option has no effect on these functions and they are still inlined.

If you want to enable automatic inlining, you use the `auto` suboption:

```
-O2 -qinline=auto
```

You can specify an inlining level 6 - 10 to achieve more aggressive automatic inlining. For example:

```
-O2 -qinline=auto:level=7
```

If automatic inlining is already enabled by default and you want to specify an inlining level of 7, you enter:

```
-O2 -qinline=level=7
```

Example 2

C

Assuming myprogram.c contains the salary, taxes, expenses, and benefits functions, you can use the following command to compile myprogram.c to inline these functions:

```
xlc myprogram.c -O2 -qinline+salary:taxes:expenses:benefits
```

If you do not want the functions salary, taxes, expenses, and benefits to be inlined, use the following command to compile myprogram.c:

```
xlc myprogram.c -O2 -qinline-salary:taxes:expenses:benefits
```

You can also disable automatic inlining and specify certain functions to be inlined with the **-qinline+** option. Consider the following example:

```
-O2 -qinline=noauto -qinline+salary:taxes:benefits
```

In this case, the functions salary, taxes, and benefits are inlined. Functions that are specified with IBM the always_inline or __always_inline__ attribute IBM or declared with the inline specifier are also inlined. No other functions are inlined.

You cannot mix the **+** and **-** suboptions with each other or with other **-qinline** suboptions. For example, the following options are invalid suboption combinations:

```
-qinline+increase-decrease    // Invalid
-qinline=level=5+increase     // Invalid
```

However, you can use multiple **-qinline** options separately. See the following example:

```
-qinline+increase -qinline-decrease -qinline=noauto:level=5
```

C

C++ In C++, you can use the **-qinline+** and **-qinline-** options in the same way as in example 2; however, you must specify the mangled function names instead of the actual function names after these options. C++

Related information

- "-g" on page 80
- "-qipa" on page 109
- "-O, -qoptimize" on page 50
- "Compiler listings" on page 13
- "always_inline (IBM extension)" in the *XL C/C++ Language Reference*

-fPIC , -fpic (-qpic)

Category

Object code control

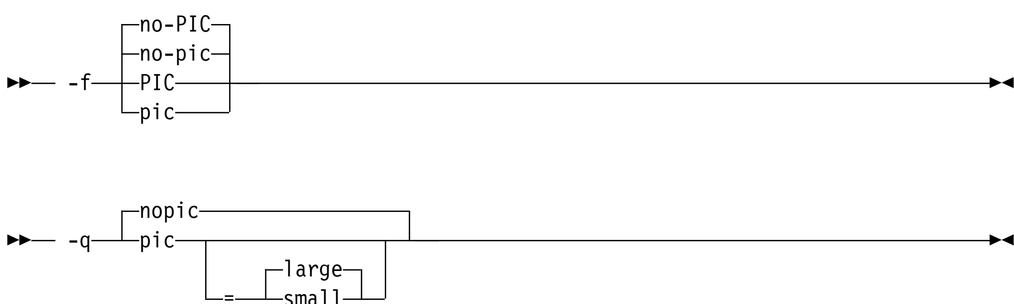
Pragma equivalent

None.

Purpose

Generates position-independent code suitable for use in shared libraries.

Syntax



Defaults

- **-fno-PIC, -fno-pic**, or **-qnopic** in both 31-bit and 64-bit compilation mode.

Specifying **-qpic** without any suboption is equivalent to **-qpic=large**.

Parameters

`small`
: Instructs the compiler to assume that the size of the Global Offset Table (GOT) is no larger than 4 Kb.

`large`
: Instructs the compiler to assume that the size of the GOT is larger than 4 Kb. Code that is generated with this option is usually larger than that generated with **-qpic=small**.

Usage

You must specify **-qpic** or **-qpic=large** when you build shared libraries.

If a thread local storage (TLS) model is not specified, the position-independent code setting determines the default TLS model:

- When **-fno-pic (-fno-PIC, -qnopic)** is in effect, the default TLS model is `local-exec`.
- When **-fPIC (-qpic)** is in effect, the default TLS model is `general-dynamic`.

If the `initial-exec` TLS model is in effect, different code sequences are used depending on different position-independent code settings.

Predefined macros

None.

Examples

To compile a shared library `libmylib.so`, use the following commands:

```
xlc mylib.c -fPIC -c -o mylib.o
xlc -shared mylib -o libmylib.so.1
```

Related information

- "-m31, -m64 (-q31, -q64)" on page 96
- "-shared (-qmkshrobj)" on page 141

-fpack-struct (-qalign)

Category

Portability and migration

Purpose

Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data.

Syntax

Defaults

-qalign=zlinux

Parameters

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

zlinux
: Uses GNU C/C++ alignment rules to maintain binary compatibility with GNU C/C++ objects.

Usage

If you use the **-fpack-struct (-qalign=bit_packed)** or **-qalign=zlinux** option more than once on the command line, the last alignment rule specified applies to the file.

Note: When using **-fpack-struct (-qalign=bit_packed)** or **-qalign=zlinux** , all system headers are also compiled with **-fpack-struct (-qalign=bit_packed)** or **-qalign=zlinux** . For a complete explanation of the option as well as usage considerations, see "Aligning data" in the *XL C/C++ Optimization and Programming Guide*.

Predefined macros

None.

Related information

- "Supported GCC pragmas" on page 160
- "Aligning data" in the *XL C/C++ Optimization and Programming Guide*
- "The aligned variable attribute" in the *XL C/C++ Language Reference*
- "The packed variable attribute" in the *XL C/C++ Language Reference*

-fsigned-bitfields, -funsigned-bitfields (-qbitfields)

Category

Floating-point and integer control

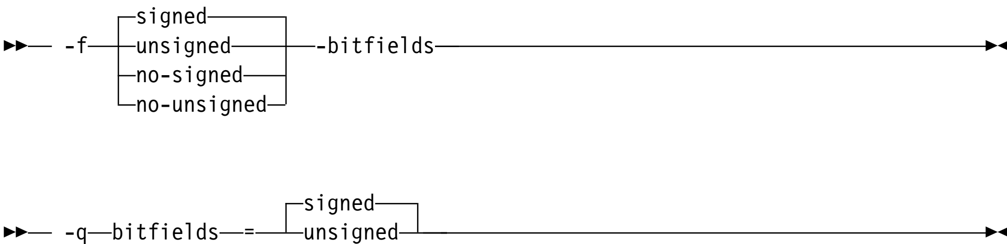
Pragma equivalent

None.

Purpose

Specifies whether bit fields are signed or unsigned.

Syntax



Defaults

-fsigned-bitfields or **-qbitfields=signed**

Parameters

signed
: Bit fields are signed.

unsigned
: Bit fields are unsigned.

Predefined macros

None.

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

Category

Floating-point and integer control

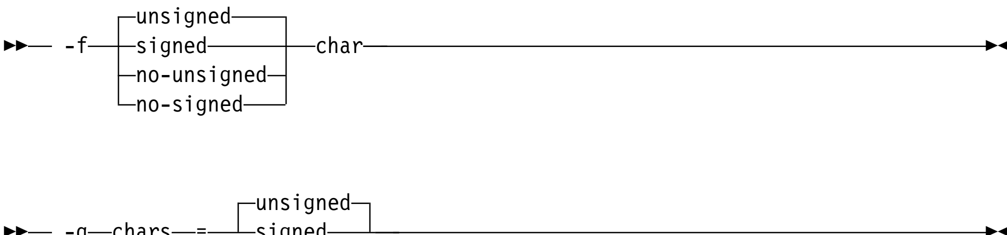
Pragma equivalent

None.

Purpose

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

Syntax



Defaults

-funsigned-char or -qchars=unsigned

Parameters

`unsigned`
: Variables of type `char` are treated as `unsigned char`.

  -fno-signed-char is equivalent to **-funsigned-char**.

`signed`
: Variables of type `char` are treated as `signed char`.

  -fno-unsigned-char is equivalent to **-fsigned-char**.

Usage

Regardless of the setting of this option or pragma, the type of `char` is still considered to be distinct from the types `unsigned char` and `signed char` for purposes of type-compatibility checking or C++ overloading.

Predefined macros

- _CHAR_SIGNED and __CHAR_SIGNED__ are defined to 1 when **signed** is in effect; otherwise, it is undefined.
- _CHAR_UNSIGNED and __CHAR_UNSIGNED__ are defined to 1 when **unsigned** is in effect; otherwise, they are undefined.

-fstrict-aliasing (-qalias=ansi), -qalias

Category

Optimization and tuning

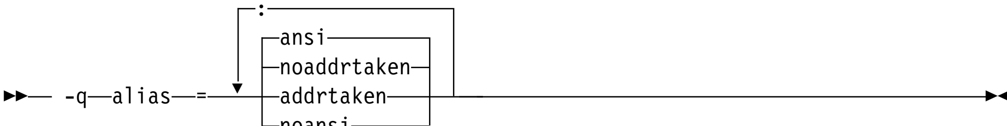
Pragma equivalent

None

Purpose

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

Syntax



For details about the `-fstrict-aliasing` option, see the GCC information, which is available at http://gcc.gnu.org/onlinedocs/.

Defaults

- C++ -qalias=noaddrtaken:ansi
- C -qalias=noaddrtaken:ansi for all invocation commands except **cc**. -qalias=noaddrtaken:noansi for the **cc** invocation command.

Parameters

addrtaken | noaddrtaken
When **addrtaken** is in effect, the reference of any variable whose address is taken may alias to any pointer type. Any class of variable for which an address has *not* been recorded in the compilation unit is considered disjoint from indirect access through pointers.

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

ansi | noansi
This suboption has no effect unless you also specify an optimization option. You can specify the `may_alias` attribute for a type that is not subject to type-based aliasing rules.

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

Usage

-qalias makes assertions to the compiler about the code that is being compiled. If the assertions about the code are false, the code that is generated by the compiler might result in unpredictable behavior when the application is run.

The following are not subject to type-based aliasing:

- Signed and unsigned types. For example, a pointer to a `signed int` can point to an `unsigned int`.
- Character pointer types can point to any type.
- Types that are qualified as `volatile` or `const`. For example, a pointer to a `const int` can point to an `int`.

- C++ Base type pointers can point to the derived types of that type. C++

Predefined macros

None.

Examples

To specify worst-case aliasing assumptions when you compile `myprogram.c`, enter:

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

Related information

- "-qipa" on page 109
- *The may_alias type attribute (IBM extension)* in the *XL C/C++ Language Reference*

-fsyntax-only (-qsyntaxonly)

Category

Error checking and debugging

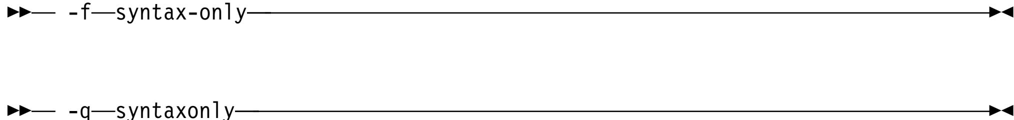
Pragma equivalent

None.

Purpose

Performs syntax checking without generating an object file.

Syntax



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:

```
xlc myprogram.c -fsyntax-only
```

Related information

- "-C, -C!" on page 43
- "-c" on page 59
- "-E" on page 45
- "-o" on page 97
- "-P" on page 52

-ftemplate-depth (-qtemplatedepth) (C++ only)

Category

Template control

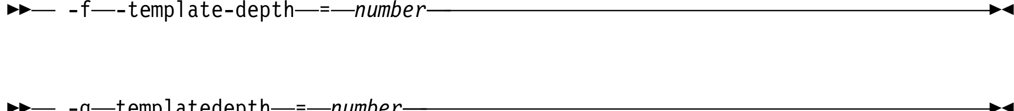
Pragma equivalent

None.

Purpose

Specifies the maximum number of recursively instantiated template specializations that will be processed by the compiler.

Syntax



Defaults

-ftemplate-depth=256 or **-qtemplatedepth=256**

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 256 is used.

Usage

Note that setting this option to a high value can potentially cause an out-of-memory error due to the complexity and amount of code generated.

Predefined macros

None.

Examples

To allow the following code in myprogram.cpp to be compiled successfully:

```
template <int n> void foo() {
  foo<n-1>();
}

template <> void foo<0>() {}

   int main() {
   foo<400>();
}
```

Enter:

```
xlc++ myprogram.cpp -ftemplate-depth=400
```

Related information

- "Using C++ templates" in the *XL C/C++ Optimization and Programming Guide*.

-ftls-model (-qtls)

Category

Object code control

Pragma equivalent

None.

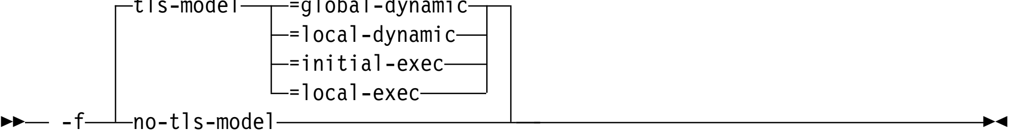
Purpose

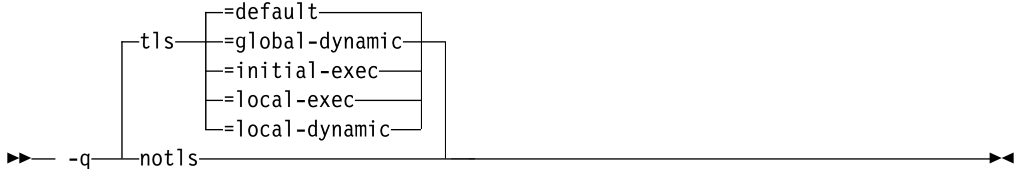
Enables recognition of the __thread storage class specifier, which designates variables that are to be allocated thread-local storage; and specifies the threadlocal storage model to be used.

When this option is in effect, any variables marked with the __thread storage class specifier are treated as local to each thread in a multithreaded application. At run time, a copy of the variable is created for each thread that accesses it, and destroyed when the thread terminates. Like other high-level constructs that you can use to parallelize your applications, thread-local storage prevents race conditions to global data, without the need for low-level synchronization of threads.

Suboptions allow you to specify thread-local storage models, which provide better performance but are more restrictive in their applicability.

Syntax





Defaults

-qtls=default

Specifying **-qtls** with no suboption is equivalent to specifying **-qtls=default**.

The default setting for **-ftls-model** is the same as the default setting for **-qtls**.

Parameters

`default (-qtls only)`
Uses the appropriate model depending on the setting of the **-fPIC (-qpic)** option, which determines whether position-independent code is generated or not. When **-fPIC (-qpic)** is in effect, this suboption results in **-qtls=global-dynamic**. When **-fno-pic (-fno-PIC, -qnopic)** is in effect, this suboption results in **-qtls=initial-exec** .

`global-dynamic`
This model is the most general, and can be used for all thread-local variables.

`initial-exec`
This model provides better performance than the global-dynamic or local-dynamic models, and can be used for thread-local variables defined in dynamically-loaded modules, provided that those modules are loaded at the same time as the executable. That is, it can only be used when all thread-local variables are defined in modules that are not loaded through `dlopen`.

`local-dynamic`
This model provides better performance than the global-dynamic model, and can be used for thread-local variables defined in dynamically-loaded modules. However, it can only be used when all references to thread-local variables are contained in the same module in which the variables are defined.

`local-exec`
This model provides the best performance of all of the models, but can only be used when all thread-local variables are defined and referenced by the main executable.

Predefined macros

None.

Related information

- "-fPIC , -fpic (-qpic)" on page 69
- "The __thread storage class specifier" in the *XL C/C++ Language Reference*

-ftime-report (-qphsinfo)

Category

Listings, messages, and compiler information

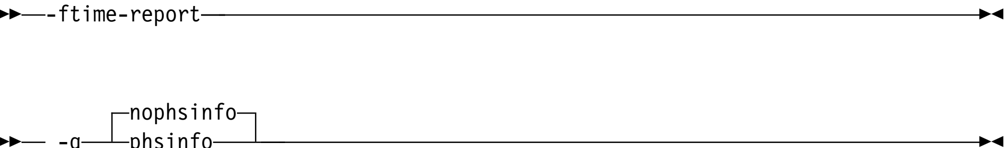
Pragma equivalent

None.

Purpose

Reports the time taken in each compilation phase to standard output.

Syntax



Defaults

-ftime-report is not on by default.

-qnophsinfo

Usage

The output takes the form *number1*/*number2* for each phase where *number1* represents the CPU time used by the compiler and *number2* represents real time (wall clock time).

The time reported by `-qphsinfo` is in seconds.

Predefined macros

None.

Example

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

```
xlc myprogram.c -ftime-report
```

The output looks like:

```
---User Time--- --System Time-- --User+System-- ---Wall Time--- --- Name ---
0.0007 (100.0%) 0.0007 (100.0%) 0.0014 (100.0%) 0.0014 (100.0%) Clang front-end timer
0.0007 (100.0%) 0.0007 (100.0%) 0.0014 (100.0%) 0.0014 (100.0%) Total

Front End - Phase Ends; 0.000/ 0.000
Compilation Time = 0:0.001088
Gen IL Time = 0:0.000288
Optimization Time = 0:0.000264
Code Gen Time = 0:0.000528
```

-ftree-vectorize (-qsimd)

Category

Optimization and tuning

Pragma equivalent

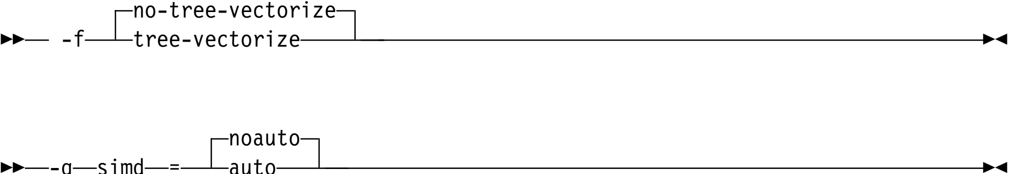
#pragma nosimd

Purpose

Controls whether the compiler can automatically take advantage of vector instructions for processors that support them.

These instructions can offer higher performance when used with algorithmic-intensive tasks such as multimedia applications.

Syntax



Defaults

-fno-tree-vectorize

-qsimd=noauto

Usage

The **-ftree-vectorize** or **-qsimd=auto** option enables automatic generation of vector instructions for processors that support them. When **-ftree-vectorize** or **-qsimd=auto** is in effect, the compiler converts certain operations that are performed in a loop on successive elements of an array into vector instructions. These instructions calculate several results at one time, which is faster than calculating each result sequentially. These options are useful for applications with significant image processing demands.

The **-fno-tree-vectorize** or **-qsimd=noauto** option disables the conversion of loop array operations into vector instructions.

Notes:
- Specifying **-qsimd** without any suboption is equivalent to **-qsimd=auto**.
- Specifying **-ftree-vectorize** or **-qsimd=auto** does not guarantee that autosimdization will occur.
- Using vector instructions to calculate several results at one time might delay or even miss detection of floating-point exceptions on some architectures. If detecting exceptions is important, do not use **-ftree-vectorize** or **-qsimd=auto**.

Rules

When you specify the **-ftree-vectorize** or **-qsimd=auto** option, the option takes effect only when the following conditions are satisfied:
- **-march** or **-qarch** is set to **z13** (equivalent to **arch11**) or higher; otherwise, the compiler ignores the option and issues a warning message.

- **-qhot** is set to **-qhot=level=0** or higher.
- The compiler runs on Linux distributions that have vector support.

If you enable IPA and specify **-ftree-vectorize** or **-qsimd=auto** at the IPA compile step, but specify **-fno-tree-vectorize** or **-qsimd=noauto** at the IPA link step, the compiler automatically sets **-ftree-vectorize** or **-qsimd=auto** at the IPA link step. It also sets an appropriate value for **-qarch** to match the architecture that is specified at the compile time. Similarly, if you enable IPA and specify **-fno-tree-vectorize** or **-qsimd=noauto** at the IPA compile step, but specify **-ftree-vectorize** or **-qsimd=auto** at the IPA link step, the compiler automatically sets **-ftree-vectorize** or **-qsimd=auto** at the compile step.

Predefined macros

None.

Examples

Any of the following command combinations can enable autosimdization when the compiler runs on a Linux distribution that has vector support:

- **xlc -O3 -qsimd -march=z13**
- **xlc -O2 -qhot=level=0 -qsimd=auto -march=z13**
- **xlc -O3 -ftree-vectorize -march=z13**

Neither of the following command combinations enables autosimdization:

- **xlc -O2 -qsimd=auto**
- **xlc -O3 -qsimd=auto -fno-tree-vectorize**

In the following example, `#pragma nosimd` is used to disable **-qsimd=auto** for a specific `for` loop:

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

Related information

- "#pragma nosimd" on page 163
- "-march (-qarch)" on page 92
- "-qreport" on page 130
- *Using interprocedural analysis* in the *XL C/C++ Optimization and Programming Guide*.

-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.

You can use different **-g** levels to balance between debug capability and compiler optimization. Higher **-g** levels provide a more complete debug support, at the cost of runtime or possible compile-time performance, while lower **-g** levels provide higher runtime performance, at the cost of some capability in the debugging session.

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

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

Syntax

Defaults

-g0

Parameters

-g

- When no optimization is enabled (**-qnoopt**), **-g** is equivalent to **-g9**.
- When the **-O2** optimization level is in effect, **-g** is equivalent to **-g2**.

-g0 Generates no debugging information. No program state is preserved.

-g1 Generates minimal read-only debugging information about line numbers and source file names. No program state is preserved. This option is equivalent to **-qlinedebug**.

-g2 Generates read-only debugging information about line numbers, source file names, and variables.

When the **-O2** optimization level is in effect, no program state is preserved.

-g3, -g4
Generates read-only debugging information about line numbers, source file names, and variables.

When the **-O2** optimization level is in effect:

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

-g5, -g6, -g7
Generates read-only debugging information about line numbers, source file names, and variables.

When the **-O2** optimization level is in effect:

- Program state is available to the debugger at `if` constructs, loop constructs, function definitions, and function calls. For details, see "Usage."
- Function parameter values are available to the debugger at the beginning of each function.

-g8 Generates read-only debugging information about line numbers, source file names, and variables.

When the **-O2** optimization level is in effect:

- 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.

-g9 Generates debugging information about line numbers, source file names, and variables. You can modify the value of the variables in the debugger.

When the **-O2** optimization level is in effect:

- 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.

Usage

When no optimization is enabled, the debugging information is always available if you specify **-g2** or a higher level. When the **-O2** optimization level is in effect, the debugging information is available at selected source locations if you specify **-g5** or a higher level.

When you specify **-g8** or **-g9** with **-O2**, the debugging information is available at every source line with an executable statement.

When you specify **-g5**, **-g6**, or **-g7** with **-O2**, the debugging information is available for the following language constructs:

- `if` constructs

  The debugging information is available at the beginning of every `if` statement, namely at the line where the `if` keyword is specified. It is also available at the beginning of the next executable statement right after the `if` construct.
- Loop constructs

  The debugging information is available at the beginning of every `do`, `for`, or `while` statement, namely at the line where the `do`, `for`, or `while` keyword is specified. It is also available at the beginning of the next executable statement right after the `do`, `for`, or `while` construct.
- Function definitions

The debugging information is available at the first executable statement in the body of the function.

- Function calls

  The debugging information is available at the beginning of every statement where a user-defined function is called. It is also available at the beginning of the next executable statement right after the statement that contains the function call.

C++ Debugging information might not be generated for programs that contain namespace alias declarations. When you compile such a program with the **-g** option, an error might be issued. You can either remove the namespace alias declarations from the program or compile the program without the **-g** option. C++

Examples

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

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

The following command uses a specific **-g** level with **-O2** to compile `myprogram.c` and generate debugging information:

```
xlc myprogram.c -O2 -g8
```

Related information

- "-qlinedebug" on page 118
- "-qfullpath" on page 103
- "-O, -qoptimize" on page 50
- "-qkeepparm" on page 115

-gdwarf (-qdbgfmt)

Category

Error checking and debugging

Pragma equivalent

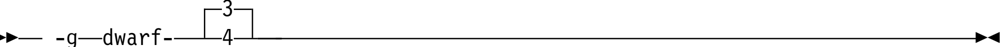
None.

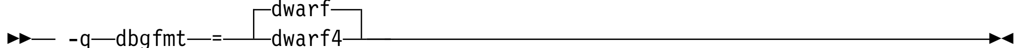
Purpose

Specifies the format for the debugging information in object files.

DWARF is a standard that defines the format of debugging information in programs. It is used on a wide variety of operating systems and is extensible and compact.

Syntax





Defaults

-qdbgfmt=dwarf (-gdwarf-3)

Parameters

dwarf
Generates debugging information in DWARF 3 format.

dwarf4
Generates debugging information in DWARF 4 format.

Usage

To debug programs built with **-qdbgfmt=dwarf** or **-qdbgfmt=dwarf4**, a DWARF-enabled debugger such as **dbx** is required.

-qdbgfmt=dwarf is equivalent to **-gdwarf-3**. **-qdbgfmt=dwarf4** is equivalent to **-gdwarf-4**.

-qdbgfmt does not imply any of the debugging options, such as **"-g" on page 80**. To generate debugging information, you must specify a debugging option, for example:

- To generate debugging information in DWARF 3 format, use **-g -qdbgfmt=dwarf**.
- To generate debugging information in DWARF 4 format, use **-g -qdbgfmt=dwarf4**.

Related information

- "-g" on page 80
- "-std (-qlanglvl)" on page 144

-include (-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 **-include (-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:

```
xlc -include test1.h -include test2.h test.c
```

Related information

- “Directory search sequence for included files” on page 9

-isystem (-qc_stdinc) (C only)

Category

Compiler customization

Pragma equivalent

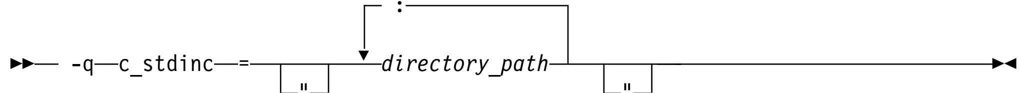
None.

Purpose

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

Syntax

```
►►──-isystem──dir──────────────────────────────────────────────►◄
```



Defaults

By default, the compiler searches the directory specified in the configuration file for the XL C header files (this is normally /opt/ibm/xlC/1.2.0/include/).

Parameters

dir
: The directory for the compiler to search for XL C header files. The search directories are after all directories specified by the `-I` option but before the standard system directories. The *dir* can be a relative or absolute path.

directory_path
: The path for the directory where the compiler should search for the XL C header files. The *directory_path* can be a relative or absolute path. You can surround the path with quotation marks to ensure it is not split up by the command line.

Usage

This option allows you to change the search paths for specific compilations. To permanently change the default search paths for the XL C headers, you use a configuration file to do so; see "Directory search sequence for included files" on page 9 for more information.

If this option is specified more than once, only the last instance of the option is used by the compiler.

This option is ignored if the **-nostdinc** or **-nostdinc++** (**-qnostdinc**) option is in effect.

Predefined macros

None.

Examples

To override the default search path for the XL C headers with `mypath/headers1` and `mypath/headers2`, enter:

```
xlc myprogram.c -isystem mypath/headers1 -isystem mypath/headers2
```

Related information

- "-isystem (-qgcc_c_stdinc) (C only)" on page 88
- "-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 137
- "-include (-qinclude)" on page 84
- "Directory search sequence for included files" on page 9
- "Specifying compiler options in a configuration file" on page 5
- "-I" on page 48

-isystem (-qcpp_stdinc) (C++ only)

Category

Compiler customization

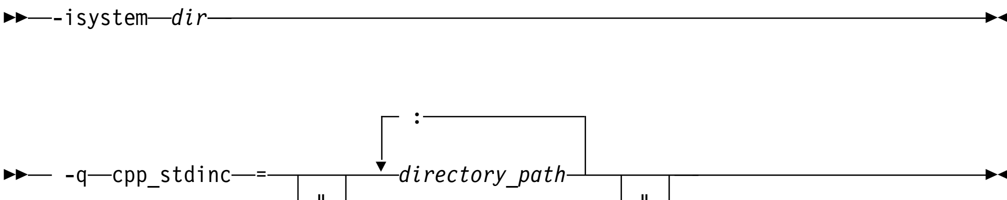
Pragma equivalent

None.

Purpose

Changes the standard search location for the XL C++ header files.

Syntax



Defaults

By default, the compiler searches the directory specified in the configuration file for the XL C++ header files (this is normally /opt/ibm/xlC/1.2.0/include/).

Parameters

dir
: The directory for the compiler to search for XL C++ header files. The search directories are after all directories specified by the `-I` option but before the standard system directories. The *dir* can be a relative or absolute path.

directory_path
: The path for the directory where the compiler should search for the XL C++ header files. The *directory_path* can be a relative or absolute path. You can surround the path with quotation marks to ensure it is not split up by the command line.

Usage

This option allows you to change the search paths for specific compilations. To permanently change the default search paths for the XL C++ headers, you use a configuration file to do so; see "Directory search sequence for included files" on page 9 for more information.

If this option is specified more than once, only the last instance of the option is used by the compiler.

This option is ignored if the **-nostdinc** or **-nostdinc++** (**-qnostdinc**) option is in effect.

Predefined macros

None.

Examples

To override the default search path for the XL C++ headers with `mypath/headers1` and `mypath/headers2`, enter:

```
xlc myprogram.C -isystem mypath/headers1 -isystem mypath/headers2
```

Related information

- "-isystem (-qgcc_cpp_stdinc) (C++ only)" on page 89
- "-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 137
- "-include (-qinclude)" on page 84
- "Directory search sequence for included files" on page 9
- "Specifying compiler options in a configuration file" on page 5
- "-I" on page 48

-isystem (-qgcc_c_stdinc) (C only)

Category

Compiler customization

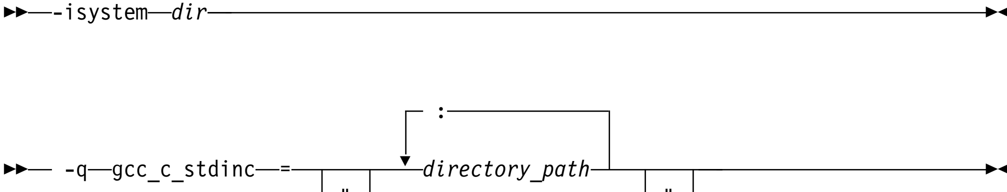
Pragma equivalent

None.

Purpose

Changes the standard search location for the GNU C system header files.

Syntax



Defaults

By default, the compiler searches the directory specified in the configuration file.

Parameters

dir
: The directory for the compiler to search for GNU C header files. The search directories are after all directories specified by the `-I` option but before the standard system directories. The *dir* can be a relative or absolute path.

directory_path
: The path for the directory where the compiler should search for the GNU C

header files. The *directory_path* can be a relative or absolute path. You can surround the path with quotation marks to ensure it is not split up by the command line.

Usage

This option allows you to change the search paths for specific compilations. To permanently change the default search paths for the GNU C headers, you use a configuration file to do so; see "Directory search sequence for included files" on page 9 for more information.

If this option is specified more than once, only the last instance of the option is used by the compiler.

This option is ignored if the **-nostdinc** or **-nostdinc++** (**-qnostdinc**) option is in effect.

Predefined macros

None.

Examples

To override the default search paths for the GNU C headers with `mypath/headers1` and `mypath/headers2`, enter:

```
xlc myprogram.c -isystem mypath/headers1 -isystem mypath/headers2
```

Related information

- "-isystem (-qc_stdinc) (C only)" on page 85
- "-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 137
- "-include (-qinclude)" on page 84
- "Directory search sequence for included files" on page 9
- "Specifying compiler options in a configuration file" on page 5
- "-I" on page 48

-isystem (-qgcc_cpp_stdinc) (C++ only)

Category

Compiler customization

Pragma equivalent

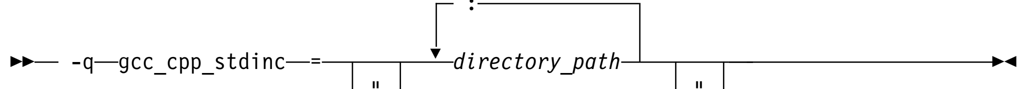
None

Purpose

Changes the standard search location for the GNU C++ system header files.

Syntax

```
►►—-isystem—dir————————————————————————————————————————►◄
```



Defaults

By default, the compiler searches the directory specified in the configuration file.

Parameters

dir
: The directory for the compiler to search for GNU C++ header files. The search directories are after all directories specified by the `-I` option but before the standard system directories. The *dir* can be a relative or absolute path.

directory_path
: The path for the directory where the compiler should search for the GNU C++ header files. The *directory_path* can be a relative or absolute path. You can surround the path with quotation marks to ensure it is not split up by the command line.

Usage

This option allows you to change the search paths for specific compilations. To permanently change the default search paths for the GNU C++ headers, you use a configuration file to do so; see "Directory search sequence for included files" on page 9 for more information.

If this option is specified more than once, only the last instance of the option is used by the compiler.

This option is ignored if the **-nostdinc** or **-nostdinc++** (**-qnostdinc**) option is in effect.

Predefined macros

None.

Examples

To override the default search paths for the GNU C++ headers with `mypath/headers1` and `mypath/headers2`, enter:

```
xlc myprogram.C -isystem mypath/headers1 -isystem mypath/headers2
```

Related information

- "-isystem (-qcpp_stdinc) (C++ only)" on page 87
- "-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 137
- "-include (-qinclude)" on page 84
- "Directory search sequence for included files" on page 9
- "Specifying compiler options in a configuration file" on page 5
- "-I" on page 48

-l

Category

Linking

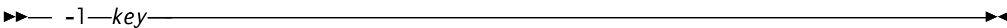
Pragma equivalent

None.

Purpose

Searches for the specified library file. The linker searches for *libkey.so*, and then *libkey.a* if *libkey.so* is not found.

Syntax



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.

For more information, refer to the **ld** documentation for your operating system.

Predefined macros

None.

Examples

To compile `myprogram.c` and link it with library `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`.

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

Related information

- "-L" on page 49
- "Specifying compiler options in a configuration file" on page 5

-march (-qarch)

Category

Optimization and tuning

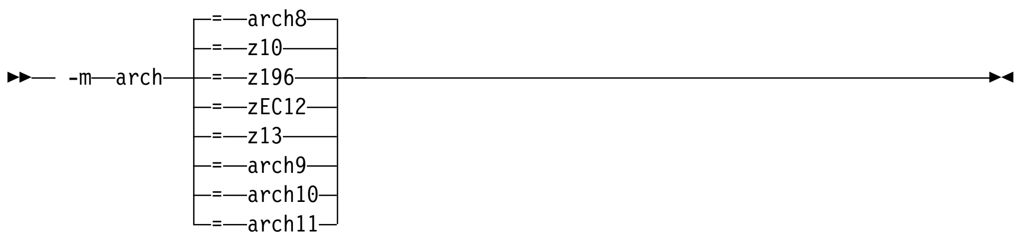
Pragma equivalent

None.

Purpose

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

Syntax



Defaults

- **-march=z10** (or **-qarch=z10**), **-march=arch8** (or **-qarch=arch8**)

Parameters

auto
: Automatically detects the specific architecture of the compilation machine. It assumes that the execution environment will be the same as the compilation environment. You can specify the **auto** suboption with **-qarch** only.

z10|arch8
: Produces code that uses instructions available on the 2097-xxx (IBM System z10® EC) models and 2098-xxx (IBM System z10 BC) models in z/Architecture® mode. z10™ EC and z10 BC machines and their follow-ons add instructions that are supported by the general instruction extensions facility, which might be exploited by the compiler.

z196|arch9
: Produces code that uses instructions available on the 2817-xxx (IBM zEnterprise® 196, also called z196) models and 2818-xxx models (IBM zEnterprise 114, also called z114) models in z/Architecture mode. z196 and z114 machines and their follow-ons add instructions that are supported by the high-word facility, the interlocked-access facility, the load- and store-on-condition facility, the distinct-operands-facility, and the population-count facility, which might be exploited by the compiler.

zEC12|arch10
: Produces code that uses instructions available on the 2827-xxx (IBM zEnterprise EC12, also called zEC12) models and 2828-xxx (IBM zEnterprise BC12, also called zBC12) models in z/Architecture mode. zEC12 and zBC12 machines and their follow-ons add instructions that are supported by the execution-hint facility, the load-and-trap facility, the miscellaneous-instruction-extension facility, and the transactional-execution facility, which might be exploited by the compiler.

z13™|arch11
: Produces code that uses instructions available on the 2964-xxx (IBM z13) models in z/Architecture mode. z13 machines and their follow-ons add instructions that are supported by the vector facility for z/Architecture, the decimal floating point packed conversion facility, the load- and store-on-condition facility, and the conditional-transaction-end facility, which might be exploited by the compiler.

For further information about these facilities, see *z/Architecture Principles of Operation* in z/OS® product documentation.

Usage

For any given **-march** or **-qarch** setting, the compiler defaults to a specific, matching **-mtune** or **-qtune** setting, which can provide additional performance improvements. For detailed information about using **-march (-qarch)** and **-mtune (-qtune)** together, see “-mtune (-qtune)” on page 94.

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

When **-qarch=auto** is specified, you must ensure that the GCC tool chain on the compilation and execution machine supports the architecture level of the machine.

Predefined macros

See “Macros related to architecture settings” on page 175 for a list of macros that are predefined by **-march (-qarch)** suboptions.

Examples

To specify that the executable program `testing` compiled from `myprogram.c` is to run on a computer with the z196 architecture, enter:

```
xlc -o testing myprogram.c -march=z196
```

Related information

- -qfloat
- “-mtune (-qtune)” on page 94

- “Specifying compiler options for architecture-specific compilation” on page 7
- “-m31, -m64 (-q31, -q64)” on page 96
- “Macros related to architecture settings” on page 175
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*

-mtune (-qtune)

Category

Optimization and tuning

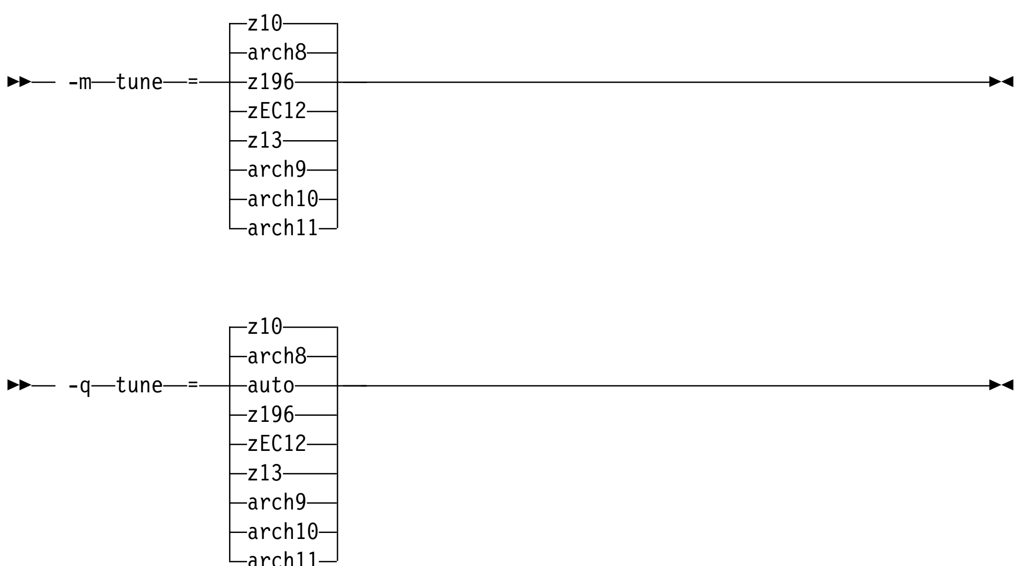
Pragma equivalent

None.

Purpose

Tunes instruction selection, scheduling, and other architecture-dependent performance enhancements to run best on a specific hardware architecture.

Syntax



Defaults

-mtune=z10 (or -qtune=z10), **-mtune=arch8 (or -qtune=arch8)**

Parameters

auto
: Optimizations are tuned for the platform on which the application is compiled. You can specify the **auto** suboption with **-qtune** only.

z10|arch8
: Optimizations are tuned for the 2097-xxx (IBM System z10 EC, also called z10 EC) models and 2098-xxx (IBM System z10 BC, also called z10 BC) models.

z196|arch9
Optimizations are tuned for the 2817-xxx (IBM zEnterprise 196, also called z196) models and 2818-xxx models (IBM zEnterprise 114, also called z114) models.

zEC12|arch10
Optimizations are tuned for the 2827-xxx (IBM zEnterprise EC12, also called zEC12) models and 2828-xxx (IBM zEnterprise BC12, also called zBC12) models.

z13|arch11
Optimizations are tuned for the 2964-xxx (IBM z13) models.

Usage

Use an **-mtune** or **-qtune** setting to match the architecture of the machine where your application runs most often. Acceptable combinations of **-march (-qarch)** and **-mtune (-qtune)** settings are shown as follows.

Table 20. Acceptable **-march** *and* **-mtune** *combinations*

-march (-qarch) option setting	Acceptable -mtune (-qtune) option settings
z10 , arch8	z10, z196, zEC12, arch8, arch9, arch10, arch11, auto
z196, arch9	z196, zEC12, arch9, arch10, arch11, auto
zEC12, arch10	zEC12, arch10, arch11, auto
z13, arch11	z13, arch11, auto

By arranging (scheduling) the generated machine instructions to take maximum advantage of hardware features such as cache size and pipelining, **-mtune** or **-qtune** can improve performance. It only has an effect when used in combination with options that enable optimization.

Although changing the **-mtune** or **-qtune** setting may affect the performance of the resulting executable, it has no effect on whether the executable can be executed correctly on a particular hardware platform.

When **-qtune=auto** is specified, you must ensure that the GCC tool chain on the compilation and execution machine supports the architecture level of the machine.

Predefined macros

None.

Examples

To specify that the executable program `testing` compiled from `myprogram.c` is to be optimized for a z196 hardware platform, enter:

```
xlc -o testing myprogram.c -mtune=z196
```

Related information

- "-march (-qarch)" on page 92
- "-m31, -m64 (-q31, -q64)" on page 96
- "Specifying compiler options for architecture-specific compilation" on page 7

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

-mzvector

Category

Language element control

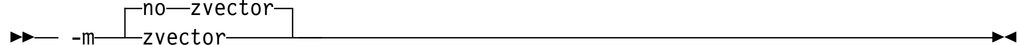
Pragma equivalent

None.

Purpose

Enables the compiler support for vector programming including the `vector` and `__vector` keywords and vector built-in functions.

Syntax



Defaults

-mno-zvector

Usage

The **-mzvector** option takes effect only on the Linux distributions that have vector support and run on the IBM z13 models. Therefore, you must specify the **-mzvector** option with the **-march=z13** option or its equivalent in effect. Otherwise, the compiler ignores the **-mzvector** option and issues an error message.

For details about **-mzvector**, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

Predefined macros

__VEC__ is defined to 10301 when **-mzvector** is in effect; otherwise, it is undefined.

Related information in the *XL C/C++ Compiler Reference*

-march (-qarch)

-ftree-vectorize (-qsimd)

Related information in the *XL C/C++ Optimization and Programming Guide*

Using vector programming support

-m31, -m64 (-q31, -q64)

Category

Object code control

Pragma equivalent

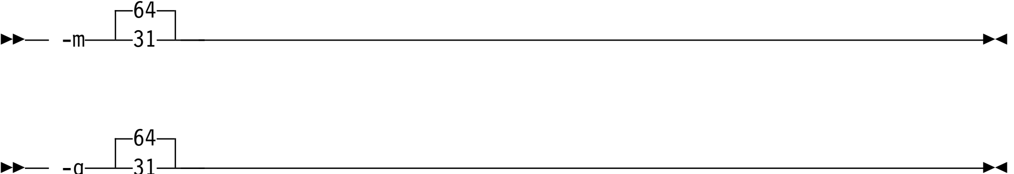
None.

Purpose

Selects either 31-bit or 64-bit compiler mode.

Use the **-m31** and **-m64** options, along with the **-march** and **-mtune** compiler options, to optimize the output of the compiler to the architecture on which that output will be used.

Syntax



Defaults

-m64 or **-q64**

Predefined macros

When **-m64 (or -q64)** is in effect, __64BIT__ is defined to 1; otherwise, it is undefined.

Examples

To specify that the executable program `testing` compiled from `myprogram.c` is to run on a computer with a 31-bit z196 architecture, enter:

```
xlc -o testing myprogram.c -m31 -march=z196
```

Related information

- Specifying compiler options for architecture-specific compilation
- “-march (-qarch)” on page 92
- “-mtune (-qtune)” on page 94

-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 4 for the default file names and suffixes produced by different phases of compilation.

Parameters

path
: When you are using the option to compile from source files, *path* can be the name of a file. *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 linker 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 **-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:

```
xlc myprogram.c -o myaccount
```

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

```
xlc test.c -c -o new.o
```

Related information

- "-c" on page 59
- "-E" on page 45
- "-P" on page 52
- "-fsyntax-only (-qsyntaxonly)" on page 74

-p, -pg, -qprofile

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Prepares the object files produced by the compiler for profiling.

When you compile with a profiling option, the compiler produces monitoring code that counts the number of times each routine is called. The compiler replaces the startup routine of each subprogram with one that calls the monitor subroutine at the start. When you execute the compiled program and it ends normally, it writes the recorded information to a gmon.out file. You can then use the **gprof** command to generate a runtime profile.

Syntax

Defaults

Not applicable.

Usage

When you are compiling and linking in separate steps, you must specify the profiling option in both steps.

The profiling options will generate full traceback tables.

Predefined macros

None.

Examples

To compile `myprogram.c` to include profiling data, enter:

```
xlc myprogram.c -p
```

Remember to compile *and* link with one of the profiling options. For example:

```
xlc myprogram.c -p -c
xlc myprogram.o -p -o program
```

Related information

- See your operating system documentation for more information on the **gprof** command.
- For details about the GCC options **-p** and **-pg**, see the GCC online documentation at http://gcc.gnu.org/onlinedocs/.

-qasm_as

Category

Compiler customization

Pragma equivalent

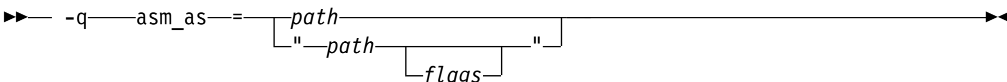
None.

Purpose

Specifies the path and flags used to invoke the assembler in order to handle assembler code in an asm assembly statement.

Normally the compiler reads the location of the assembler from the configuration file; you can use this option to specify an alternate assembler program and flags to pass to that assembler.

Syntax



Defaults

By default, the compiler invokes the assembler program defined for the **as** command in the compiler configuration file.

Parameters

path
The full path name of the assembler to be used.

flags
A space-separated list of options to be passed to the assembler for assembly statements. Quotation marks must be used if spaces are present.

Predefined macros

None.

Examples

To instruct the compiler to use the assembler program at /bin/as when it encounters inline assembler code in myprogram.c, enter the following command:

```
xlc myprogram.c -qasm_as=/bin/as
```

To instruct the compiler to pass some additional options to the assembler at /bin/as for processing inline assembler code in myprogram.c, enter the following command:

```
xlc myprogram.c -qasm_as="/bin/as -a64 -l a.lst"
```

Related information

- "-fasm (-qasm)" on page 61

-qcrt, -nostartfiles (-qnocrt)

Category

Linking

Pragma equivalent

None.

Purpose

When **-qcrt** is in effect, the system startup routines are automatically linked. When **-nostartfiles (-qnocrt)** is in effect, the system startup files are not used at link time; only the files specified on the command line with the **-l** flag are linked.

This option can be used in system programming to disable the automatic linking of the startup routines provided by the operating system.

Syntax

Defaults

-qcrt

Predefined macros

None.

Related information

- "-qlib, -nodefaultlibs (-qnolib)" on page 116

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

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

Predefined macros

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

Related information

- "-qrtti, -fno-rtti (-qnortti) (C++ only)" on page 131
- The **-fexceptions** option that GCC provides. For details, see the GCC online documentation at http://gcc.gnu.org/onlinedocs/.

-qfloat

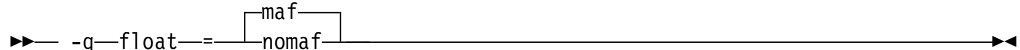
Category

Floating-point and integer control

Purpose

Selects different strategies for speeding up or improving the accuracy of floating-point calculations.

Syntax



Defaults

- **-qfloat=maf**

Parameters

maf | nomaf
Makes floating-point calculations faster and more accurate by using floating-point multiply-add instructions appropriately. The results might not be equivalent to those from similar calculations performed at compile time or on other types of computers. Negative zero results might be produced. This suboption might affect the precision of floating-point results in the intermediate calculation.

If **-qfloat=nomaf** is specified, the multiply-add instructions are generated only if they are required for correctness.

Usage

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.

Predefined macros

None.

Example

To compile the source file `myprogram.c` without multiply-add instructions generated, run the following command:

```
xlc myprogram.c -qfloat=nomaf
```

Related information

- "-march (-qarch)" on page 92
- "Handling floating-point operations" in the *XL C/C++ Optimization and Programming Guide*

-qfullpath

Category

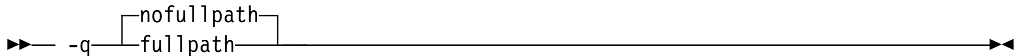
Error checking and debugging

Purpose

When used with the **-g** or **-qlinedebug** option, this option records the full, or absolute, path names of source and include files in object files compiled with debugging information, so that debugging tools can correctly locate the source files.

When **fullpath** is in effect, the absolute (full) path names of source files are preserved. When **nofullpath** is in effect, the relative path names of source files are preserved.

Syntax



Defaults

-qnofullpath

Usage

If your executable file was moved to another directory, the debugger would be unable to find the file unless you provide a search path in the debugger. You can use **fullpath** to ensure that the debugger locates the file successfully.

Predefined macros

None.

Related information

- "-qlinedebug" on page 118
- "-g" on page 80

-qfuncsect

Category

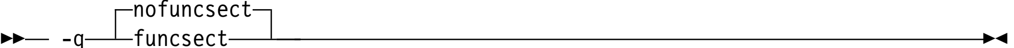
Object code control

Purpose

Places instructions for each function in a separate section. Placing each function in its own section might reduce the size of your program because the linker can collect garbage per function rather than per object file.

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

Syntax



Defaults

-qnofuncsect

Usage

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

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

Predefined macros

None.

-qhot

Category

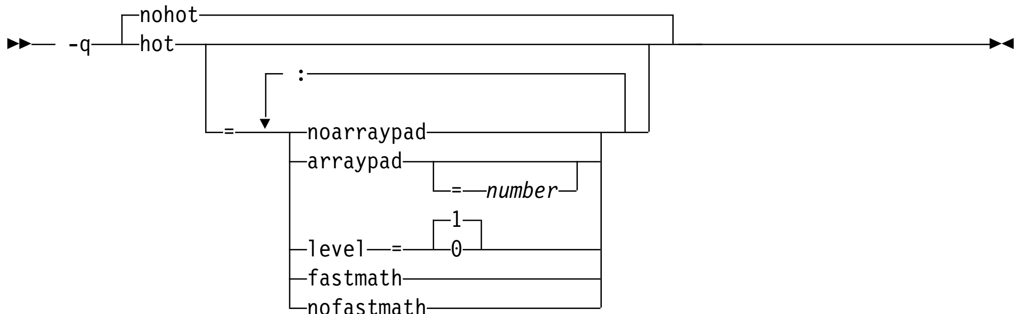
Optimization and tuning

Purpose

Performs high-order loop analysis and transformations (HOT) during optimization.

The **-qhot** compiler option is a powerful alternative to hand tuning that provides opportunities to optimize loops and array language. This compiler option will always attempt to optimize loops, regardless of the suboptions you specify.

Syntax



Defaults

- **-qnohot**
- **-qhot=noarraypad:level=0:fastmath** when **-O3** is in effect.
- Specifying **-qhot** without suboptions is equivalent to **-qhot=noarraypad:level=1:fastmath**.

Parameters

arraypad | noarraypad

Permits the compiler to increase the dimensions of arrays where doing so might improve the efficiency of array-processing loops. (Because of the implementation of the cache architecture, array dimensions that are powers of two can lead to decreased cache utilization.) Specifying **-qhot=arraypad** when your source includes large arrays with dimensions that are powers of 2 can reduce cache misses and page faults that slow your array processing programs. This can be particularly effective when the first dimension is a power of 2. If you use this suboption with no *number*, the compiler will pad any arrays where it infers there may be a benefit and will pad by whatever amount it chooses. Not all arrays will necessarily be padded, and different arrays may be padded by different amounts. If you specify a *number*, the compiler will pad every array in the code.

Note: Using **arraypad** can be unsafe, as it does not perform any checking for reshaping or equivalences that may cause the code to break if padding takes place.

number

A positive integer value representing the number of elements by which each array will be padded in the source. The pad amount must be a positive integer value. To achieve more efficient cache utilization, it is recommended that pad values be multiples of the largest array element size, typically 4, 8, or 16.

level=0

Performs a subset of the high-order transformations and sets the default to **noarraypad:fastmath**.

level=1

Performs the default set of high-order transformations.

fastmath | nofastmath

You can use this suboption to tune your application to either use fast scalar versions of math functions or use the default versions.

-qhot=nofastmath disables the replacement of math routines by the XLOPT library. **-qhot=fastmath** is enabled by default if **-qhot** is specified regardless of the hot level.

Usage

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

You can use the **-qreport** option in conjunction with **-qhot** or any optimization option that implies **-qhot** to produce a pseudo-C report showing how the loops were transformed. The loop transformations are included in the listing report if is also specified. For more information, see "-qreport" on page 130.

Predefined macros

None.

Related information

- "-march (-qarch)" on page 92
- "-ftree-vectorize (-qsimd)" on page 78
- "-qreport" on page 130
- "-O, -qoptimize" on page 50

- *Using the Mathematical Acceleration Subsystem (MASS)* in the *XL C/C++ Optimization and Programming Guide*

-qinitauto

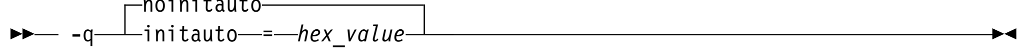
Category

Error checking and debugging

Purpose

Initializes uninitialized automatic variables to a specific value, for debugging purposes.

Syntax



Defaults

-qnoinitauto

Parameters

hex_value
A one- to eight-digit hexadecimal number.

- To initialize each byte of storage to a specific value, specify one or two digits for the *hex_value*.
- To initialize each word of storage to a specific value, specify three to eight digits for the *hex_value*.

- In the case where less than the maximum number of digits are specified for the size of the initializer requested, leading zeros are assumed.
- In the case of word initialization, if an automatic variable is smaller than a multiple of 4 bytes in length, the *hex_value* is truncated on the left to fit. For example, if an automatic variable is only 1 byte and you specify five digits for the *hex_value*, the compiler truncates the three digits on the left and assigns the other two digits on the right to the variable. See Example 1.
- If an automatic variable is larger than the *hex_value* in length, the compiler repeats the *hex_value* and assigns it to the variable. See Example 1.
- If the automatic variable is an array, the *hex_value* is copied into the memory location of the array in a repeating pattern, beginning at the first memory location of the array. See Example 2.
- You can specify alphabetic digits as either uppercase or lowercase.
- The *hex_value* can be optionally prefixed with `0x`, in which `x` is case-insensitive.

Usage

The -qinitauto option provides the following benefits:
- Setting *hex_value* to zero ensures that all automatic variables that are not explicitly initialized when declared are cleared before they are used.
- You can use this option to initialize variables of real or complex type to a signaling or quiet NaN, which helps locate uninitialized variables in your program.

This option generates extra code to initialize the value of automatic variables. It reduces the runtime performance of the program and is to be used for debugging purposes only.

Restrictions:
- Objects that are equivalenced, structure components, and array elements are not initialized individually. Instead, the entire storage sequence is initialized collectively.
- The **-qinitauto=hex_value** option does not initialize variable length arrays or memory allocated through the `__alloca` function.

Predefined macros

- __INITAUTO__ is defined to the least significant byte of the *hex_value* that is specified on the **-qinitauto** option or pragma; otherwise, it is undefined.
- __INITAUTO_W__ is defined to the byte *hex_value*, repeated four times, or to the word *hex_value*, which is specified on the **-qinitauto** option or pragma; otherwise, it is undefined.

For example:
- For option -qinitauto=0xABCD, the value of __INITAUTO__ is 0xCDu, and the value of __INITAUTO_W__ is 0x0000ABCDu.
- For option -qinitauto=0xCD, the value of __INITAUTO__ is 0xCDu, and the value of __INITAUTO_W__ is 0xCDCDCDCDu.

Examples

Example 1: Use the **-qinitauto** option to initialize automatic variables of scalar types.

```
#include <stdio.h>

int main()
{
  char a;
  short b;
  int c;
  long long int d;

  printf("char a = 0x%X\n",(char)a);
  printf("short b = 0x%X\n",(short)b);
  printf("int c = 0x%X\n",c);
  printf("long long int d = 0x%llX\n",d);
}
```

If you compile the program with `-qinitauto=AABBCCDD`, for example, the result is as follows:

```
char a = 0xDD
short b = 0xFFFFCCDD
int c = 0xAABBCCDD
long long int d = 0xAABBCCDDAABBCCDD
```

Example 2: Use the **-qinitauto** option to initialize automatic array variables.

```
#include <stdio.h>
#define ARRAY_SIZE 5

int main()
{
  char a[5];
  short b[5];
  int c[5];
  long long int d[5];

  printf("array of char: ");
  for (int i = 0; i<ARRAY_SIZE; i++)
    printf("0x%1X ",(unsigned)a[i]);
  printf("\n");

  printf("array of short: ");
  for (int i = 0; i<ARRAY_SIZE; i++)
    printf("0x%1X ",(unsigned)b[i]);
  printf("\n");

  printf("array of int: ");
  for (int i = 0; i<ARRAY_SIZE; i++)
    printf("0x%1X ",(unsigned)c[i]);
  printf("\n");

  printf("array of long long int: ");
  for (int i = 0; i<ARRAY_SIZE; i++)
    printf("0x%1X ",(unsigned)d[i]);
  printf("\n");
}
```

If you compile the program with `-qinitauto=AABBCCDD`, for example, the result is as follows:

```
array of char: 0xAA 0xBB 0xCC 0xDD 0xAA
array of short: 0xAABB 0xCCDD 0xAABB 0xCCDD 0xAABB
array of int: 0xAABBCCDD 0xAABBCCDD 0xAABBCCDD 0xAABBCCDD 0xAABBCCDD
array of long long int: 0xAABBCCDDAABBCCDD 0xAABBCCDDAABBCCDD 0xAABBCCDDAABBCCDD
0xAABBCCDDAABBCCDD 0xAABBCCDDAABBCCDD
```

-qipa

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

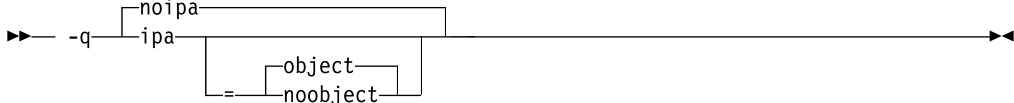
Enables or customizes a class of optimizations known as interprocedural analysis (IPA).

IPA is a two-step process: the first step, which takes place during compilation, consists of performing an initial analysis and storing interprocedural analysis information in the object file. The second step, which takes place during linking, and causes a complete recompilation of the entire application, applies the optimizations to the entire program.

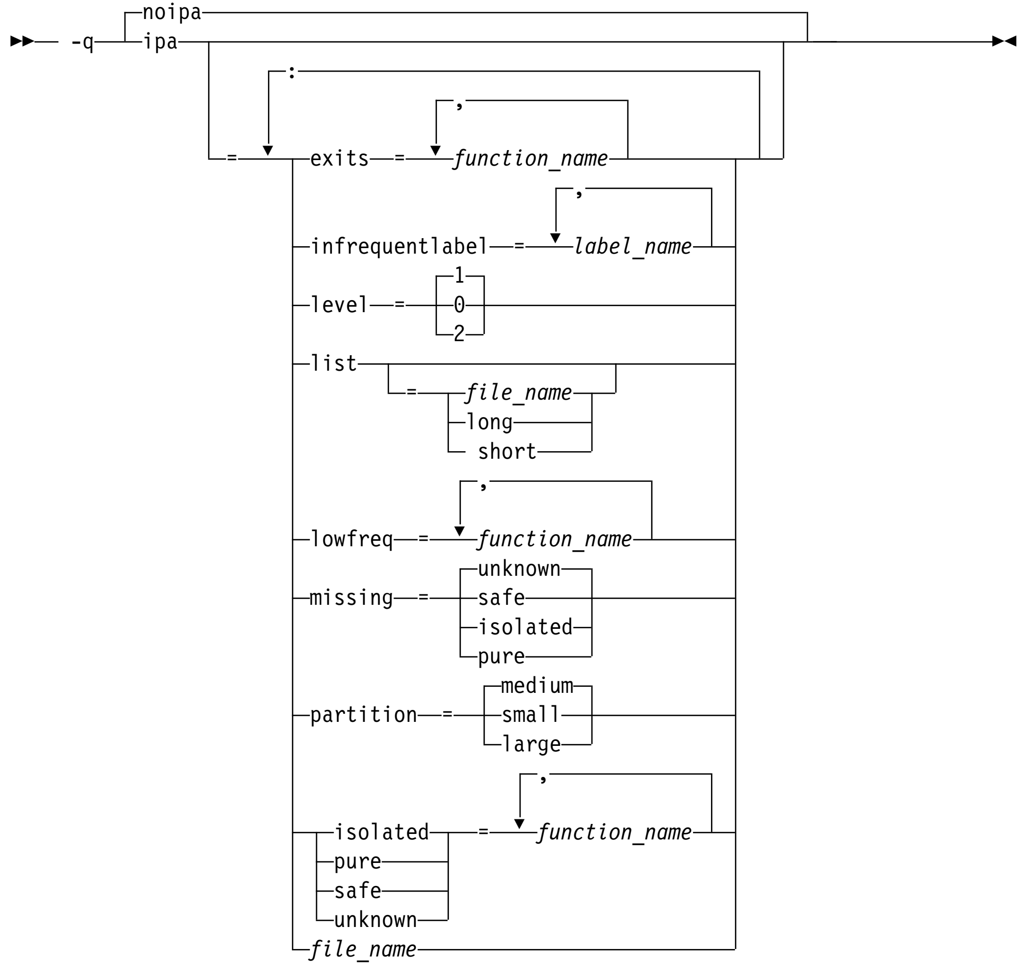
You can use **-qipa** during the compilation step, the link step, or both. If you compile and link in a single compiler invocation, only the link-time suboptions are relevant. If you compile and link in separate compiler invocations, only the compile-time suboptions are relevant during the compile step, and only the link-time suboptions are relevant during the link step.

Syntax

-qipa compile-time syntax



-qipa link-time syntax



Defaults

- **-qnoipa**

Parameters

You can specify the following parameters during a separate compile step only:

object | **noobject**
: Specifies whether to include standard object code in the output object files.

Specifying **noobject** can substantially reduce overall compile time by not generating object code during the first IPA phase. Note that if you specify **-S** with **noobject**, **noobject** will be ignored.

If compiling and linking are performed in the same step and you do not specify the **-S** or any listing option, **-qipa=noobject** is implied.

Specifying **-qipa** with no suboptions on the compile step is equivalent to **-qipa=object**.

You can specify the following parameters during a combined compilation and link stepin the same compiler invocation, or during a separate link step only:

clonearch | **noclonearch**
: This suboption is no longer supported.

cloneproc | nocloneproc
This suboption is no longer supported.

exits
Specifies names of functions which represent program exits. Program exits are calls which can never return and can never call any function which has been compiled with IPA pass 1. The compiler can optimize calls to these functions (for example, by eliminating save/restore sequences), because the calls never return to the program. These functions must not call any other parts of the program that are compiled with **-qipa**.

infrequentlabel
Specifies user-defined labels that are likely to be called infrequently during a program run.

label_name
The name of a label, or a comma-separated list of labels.

isolated
Specifies a comma-separated list of functions that are not compiled with **-qipa**. Functions that you specify as *isolated* or functions within their call chains cannot refer directly to any global variable.

level
Specifies the optimization level for interprocedural analysis. Valid suboptions are as follows:

0 Performs only minimal interprocedural analysis and optimization.

1 Enables inlining, limited alias analysis, and limited call-site tailoring.

2 Performs full interprocedural data flow and alias analysis.

If you do not specify a level, the default is 1.

To generate data reorganization information, specify the optimization level **-qipa=level=2** together with **-qreport**. During the IPA link phase, the data reorganization messages for program variable data are produced in the data reorganization section of the listing file. Reorganizations include array splitting, array transposing, memory allocation merging, array interleaving, and array coalescing.

list
Specifies that a listing file be generated during the link phase. The listing file contains information about transformations and analyses performed by IPA, as well as an optional object listing for each partition.

If you do not specify a *list_file_name*, the listing file name defaults to a.lst. If you specify **-qipa=list** together with any other option that generates a listing file, IPA generates an a.lst file that overwrites any existing a.lst file. If you have a source file named a.c, the IPA listing will overwrite the regular compiler listing a.lst. You can use the **-qipa=list=***list_file_name* suboption to specify an alternative listing file name.

Additional suboptions are one of the following suboptions:

short Requests less information in the listing file. Generates the Object File Map, Source File Map and Global Symbols Map sections of the listing.

long Requests more information in the listing file. Generates all of the

sections generated by the **short** suboption, plus the Object Resolution Warnings, Object Reference Map, Inliner Report and Partition Map sections.

lowfreq
Specifies functions that are likely to be called infrequently. These are typically error handling, trace, or initialization functions. The compiler may be able to make other parts of the program run faster by doing less optimization for calls to these functions.

missing
Specifies the interprocedural behavior of functions that are not compiled with **-qipa** and are not explicitly named in an **unknown**, **safe**, **isolated**, or **pure** suboption.

Valid suboptions are one of the following suboptions:

safe Specifies that the missing functions do not indirectly call a visible (not missing) function either through direct call or through a function pointer.

isolated
Specifies that the missing functions do not directly reference global variables accessible to visible function. Functions bound from shared libraries are assumed to be *isolated*.

pure Specifies that the missing functions are *safe* and *isolated* and do not indirectly alter storage accessible to visible functions. *pure* functions also have no observable internal state.

unknown
Specifies that the missing functions are not known to be *safe*, *isolated*, or *pure*. This suboption greatly restricts the amount of interprocedural optimization for calls to missing functions.

The default is to assume **unknown**.

partition
Specifies the size of each program partition created by IPA during pass 2. Valid suboptions are one of the following suboptions:

- **small**
- **medium**
- **large**

Larger partitions contain more functions, which result in better interprocedural analysis but require more storage to optimize. Reduce the partition size if compilation takes too long because of paging.

pure
Specifies *pure* functions that are not compiled with **-qipa**. Any function specified as *pure* must be *isolated* and *safe*, and must not alter the internal state nor have side-effects, defined as potentially altering any data visible to the caller.

safe
Specifies *safe* functions that are not compiled with **-qipa** and do not call any other part of the program. Safe functions can modify global variables, but may not call functions compiled with **-qipa**.

unknown
Specifies *unknown* functions that are not compiled with **-qipa**. Any function

specified as *unknown* can make calls to other parts of the program compiled with **-qipa**, and modify global variables.

file_name
Gives the name of a file which contains suboption information in a special format.

The file format is shown as follows:

```
# ... comment
attribute{, attribute} = name{, name}
missing = attribute{, attribute}
exits = name{, name}
lowfreq = name{, name}
list [ = file-name | short | long ]
level = 0 | 1 | 2
partition = small | medium | large
```

where *attribute* is one of:
- exits
- lowfreq
- unknown
- safe
- isolated
- pure

Usage

Specifying **-qipa** automatically sets the optimization level to **-O2**. For additional performance benefits, you can also specify the **-finline-functions (-qinline)** option. The **-qipa** option extends the area that is examined during optimization and inlining from a single function to multiple functions (possibly in different source files) and the linkage between them.

If any object file used in linking with **-qipa** was created with the **-qipa=noobject** option, any file containing an entry point (the main program for an executable program, or an exported function for a library) must be compiled with **-qipa**.

You can link objects created with different releases of the compiler, but you must ensure that you use a linker that is at least at the same release level as the newer of the compilers used to create the objects being linked.

Some symbols which are clearly referenced or set in the source code may be optimized away by IPA, and may be lost to **debug** or **nm** outputs. Using IPA together with the **-g** compiler will usually result in non-steppable output.

Note that if you specify **-qipa** with **-#**, the compiler does not display linker information subsequent to the IPA link step.

For recommended procedures for using **-qipa**, see "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*.

Predefined macros

None.

Examples

The following example shows how you might compile a set of files with interprocedural analysis:

```
xlc -c *.c -qipa
xlc -o product *.o -qipa
```

Here is how you might compile the same set of files, improving the optimization of the second compilation, and the speed of the first compile step. Assume that there exist a set of routines, `user_trace1`, `user_trace2`, and `user_trace3`, which are rarely executed, and the routine `user_abort` that exits the program:

```
xlc -c *.c -qipa=noobject
xlc -c *.o -qipa=lowfreq=user_trace[123]:exit=user_abort
```

Related information

- "-finline-functions (-qinline)" on page 65
- "-qisolated_call"
- "#pragma execution_frequency" on page 162
- "-S" on page 54
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*
- Runtime environment variables

-qisolated_call

Category

Optimization and tuning

Purpose

Specifies functions in the source file that have no side effects other than those implied by their parameters.

Essentially, any change in the state of the runtime environment is considered a side effect, including:

- Accessing a volatile object
- Modifying an external object
- Modifying a static object
- Modifying a file
- Accessing a file that is modified by another process or thread
- Allocating a dynamic object, unless it is released before returning
- Releasing a dynamic object, unless it was allocated during the same invocation
- Changing system state, such as rounding mode or exception handling
- Calling a function that does any of the above

Marking a function as isolated indicates to the optimizer that external and static variables cannot be changed by the called function and that pessimistic references to storage can be deleted from the calling function where appropriate. Instructions can be reordered with more freedom, resulting in fewer pipeline delays and faster execution in the processor. Multiple calls to the same function with identical parameters can be combined, calls can be deleted if their results are not needed, and the order of calls can be changed.

Syntax

Option syntax

Defaults

Not applicable.

Parameters

function
The name of a function that does not have side effects or does not rely on functions or processes that have side effects. *function* is a primary expression that can be an identifier, operator function, conversion function, or qualified name. An identifier must be of type function or a `typedef` of function. C++ If the name refers to an overloaded function, all variants of that function are marked as isolated calls. C++

Usage

The only side effect that is allowed for a function named in the option or pragma is modifying the storage pointed to by any pointer arguments passed to the function, that is, calls by reference. The function is also permitted to examine nonvolatile external objects and return a result that depends on the nonvolatile state of the runtime environment. Do not specify a function that causes any other side effects; that calls itself; or that relies on local static storage. If a function is incorrectly identified as having no side effects, the program behavior might be unexpected or produce incorrect results.

Predefined macros

None.

Examples

To compile `myprogram.c`, specifying that the functions `myfunction(int)` and `classfunction(double)` do not have side effects, enter:

```
xlc myprogram.c -qisolated_call=myfunction:classfunction
```

Related information

- "The const function attribute" and "The pure function attribute" in the *XL C/C++ Language Reference*

-qkeepparm

Category

Error checking and debugging

Pragma equivalent

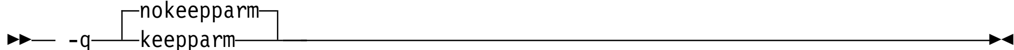
None.

Purpose

When used with **-O2** or higher optimization, specifies whether procedure parameters are stored on the stack.

A function usually stores its incoming parameters on the stack at the entry point. However, when you compile code with optimization options enabled, the compiler may remove these parameters from the stack if it sees an optimizing advantage in doing so. When **-qkeepparm** is in effect, parameters are stored on the stack even when optimization is enabled. When **-qnokeepparm** is in effect, parameters are removed from the stack if this provides an optimization advantage.

Syntax



Defaults

-qnokeepparm

Usage

Specifying **-qkeepparm** that the values of incoming parameters are available to tools, such as debuggers, by preserving those values on the stack. However, this may negatively affect application performance.

Predefined macros

None.

Related information

- "-O, -qoptimize" on page 50

-qlib, -nodefaultlibs (-qnolib)

Category

Linking

Pragma equivalent

None.

Purpose

Specifies whether standard system libraries and XL C/C++ libraries are to be linked.

When **-qlib** is in effect, the standard system libraries and compiler libraries are automatically linked. When **-nodefaultlibs (-qnolib)** is in effect, the standard system libraries and compiler libraries are not used at link time; only the libraries specified on the command line with the **-l** flag will be linked.

This option can be used in system programming to disable the automatic linking of unneeded libraries.

Syntax

Defaults

-qlib

Usage

Using **-nodefaultlibs (-qnolib)** specifies that no libraries, including the system libraries as well as the XL C/C++ libraries (these are found in the lib/ and lib64/ subdirectories of the compiler installation directory), are to be linked. The system startup files are still linked, unless **-nostartfiles (-qnocrt)** is also specified.

Note: If your program references any symbols that are defined in the standard libraries or compiler-specific libraries, link errors will occur. To avoid these unresolved references when compiling with **-nodefaultlibs (-qnolib)**, be sure to explicitly link the required libraries by using the command flag **-l** and the library name.

Predefined macros

None.

Examples

To compile `myprogram.c` without linking to any libraries except the compiler library libxlopt.a, enter:

```
xlc myprogram.c -nodefaultlibs -lxlopt
```

Related information

- "-qcrt, -nostartfiles (-qnocrt)" on page 100

-qlibansi

Category

Optimization and tuning

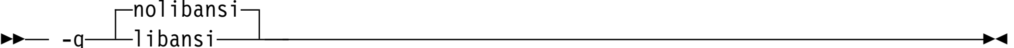
Pragma equivalent

Purpose

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

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

Syntax



Defaults

-qnolibansi

Predefined macros

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

-qlinedebug

Category

Error checking and debugging

Pragma equivalent

None.

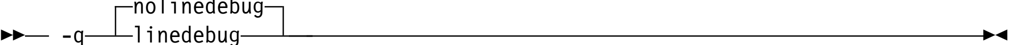
Purpose

Generates only line number and source file name information for a debugger.

When **-qlinedebug** is in effect, the compiler produces minimal debugging information, so the resulting object size is smaller than that produced by the **-g** debugging option. You can use the debugger to step through the source code, but you will not be able to see or query variable information. The traceback table, if generated, will include line numbers.

-qlinedebug is equivalent to **-g1**.

Syntax



Defaults

-qnolinedebug

Usage

When **-qlinedebug** is in effect, function inlining is disabled.

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

The **-g** option overrides the **-qlinedebug** option. If you specify **-g** with **-qnolinedebug** on the command line, **-qnolinedebug** is ignored and a warning is issued.

Predefined macros

None.

Examples

To compile `myprogram.c` to produce an executable program `testing` so you can step through it with a debugger, enter:

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

Related information

- "-g" on page 80
- "-O, -qoptimize" on page 50

-qlist

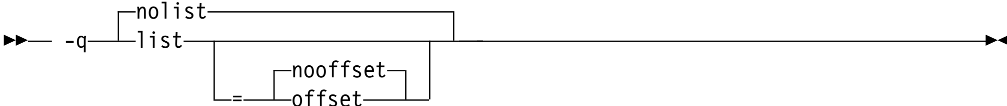
Category

Listings, messages, and compiler information

Purpose

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

Syntax



Defaults

-qnolist

Parameters

offset | **nooffset**
Changes the offset of the PDEF header from `00000` to the offset of the start of the text area. Specifying the option allows any program reading the .lst file to add the value of the PDEF and the line in question, and come up with the same value whether **offset** or **nooffset** is specified. The **offset** suboption is only relevant if there are multiple procedures in a compilation unit.

Specifying **list** without the suboption is equivalent to **list=nooffset**.

Usage

When **list** is in effect, a listing file is generated with a .lst suffix for each source file named on the command line. For details of the contents of the listing file, see "Compiler listings" on page 13.

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.

Examples

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

```
xlc myprogram.c -qlist
```

-qmakedep, -MD (-qmakedep=gcc)

Category

Output control

Pragma equivalent

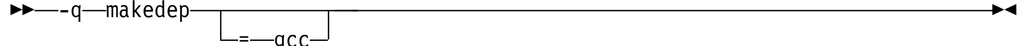
None.

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**.

If you specify **-qmakedep** with no suboption, the dependency output file specifies a separate rule for each of the main source file's dependencies.

Usage

For each source file with a .c, .C, .cpp, or .i suffix 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. For more information, see the Examples section.

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 contains a line for the input file and an entry for each include file. It has the general form:

```
file_name.o:include_file_name
file_name.o:file_name.suffix
```

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 9. 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:

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

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

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

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

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

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

```
xlc -c -qmakedep foo_src.c -o foo_obj.o
```

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

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

Related information

- "-o" on page 97
- "Directory search sequence for included files" on page 9

- The **-M**, **-MD**, **-MF**, **-MG**, **-MM**, **-MMD**, **-MP**, **-MQ**, and **-MT** options that GCC provides. For details, see the GCC online documentation at http://gcc.gnu.org/onlinedocs/.

-qpath

Category

Compiler customization

Pragma equivalent

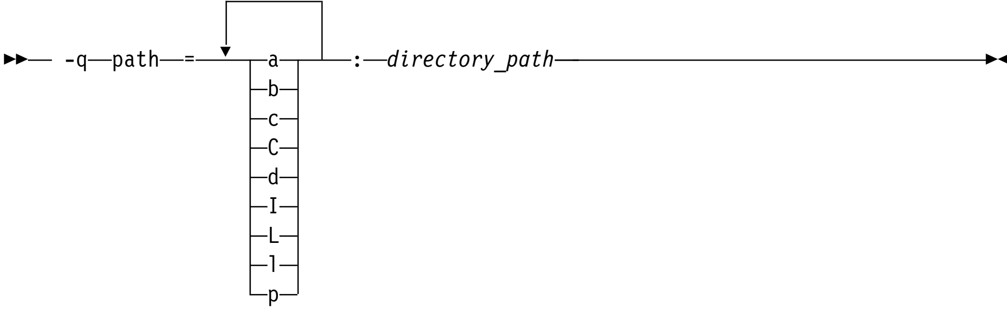
None.

Purpose

Specifies substitute path names for XL C/C++ components such as the compiler, assembler, linker, and preprocessor.

You can use this option if you want to keep multiple levels of some or all of the XL C/C++ components and have the option of specifying which one you want to use. This option is preferred over the **-B** and **-t** options.

Syntax



Defaults

By default, the compiler uses the paths for compiler components defined in the configuration file.

Parameters

directory_path
The path to the directory where the alternate programs are located.

The following table shows the correspondence between **-qpath** parameters and the component names:

Parameter	Description	Component name
a	The assembler	as
b	The low-level optimizer	xlCcode
c, C	The C and C++ compiler front end	xlCentry

Parameter	Description	Component name
d	The disassembler	dis
I (uppercase i)	The high-level optimizer, compile step	ipa
L	The high-level optimizer, link step	ipa
l (lowercase L)	The linker	ld
p	The preprocessor	xlCentry

Usage

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

Predefined macros

None.

Examples

To compile `myprogram.c` using a substitute **xlc** compiler in `/lib/tmp/mine/`, enter the command:

```
xlc myprogram.c -qpath=c:/lib/tmp/mine/
```

To compile `myprogram.c` using a substitute linker in `/lib/tmp/mine/`, enter the command:

```
xlc myprogram.c -qpath=l:/lib/tmp/mine/
```

Related information

- “-B” on page 42
- “-F” on page 46
- “-t” on page 148

-qpdf1, -qpdf2

Category

Optimization and tuning

Pragma equivalent

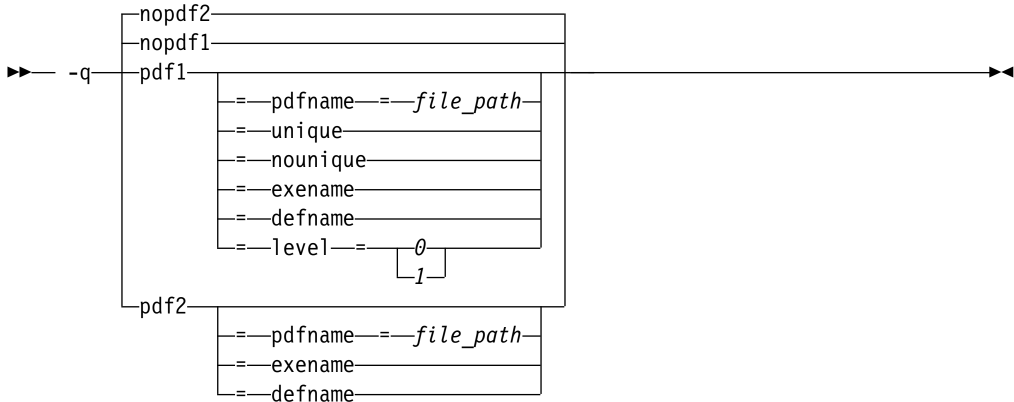
None.

Purpose

Tunes optimizations through *profile-directed feedback* (PDF), where results from sample program execution are used to improve optimization near conditional branches and in frequently executed code sections.

Optimizes an application for a typical usage scenario based on an analysis of how often branches are taken and blocks of code are run.

Syntax



Defaults

-qnopdf1, -qnopdf2

Parameters

defname
Reverts a PDF file to its default file name if the **-qpdf1=exename** option is also specified.

exename
Specifies the name of the generated PDF file according to the output file name specified by the **-o** option. For example, you can use **-qpdf1=exename -o func func.c** to generate a PDF file called `.func_pdf`.

level=0 | 1
Specifies different levels of profiling information to be generated by the resulting application. The following table shows the type of profiling information supported on each level. The plus sign (+) indicates that the profiling type is supported.

Table 21. Profiling type supported on each -qpdf1 level

Profiling type	Level	
	0	1
Block-counter profiling	+	+
Call-counter profiling	+	+
Value profiling		+

-qpdf1=level=1 is the default level. It is equivalent to **-qpdf1**. Higher PDF levels profile more optimization opportunities but have a larger overhead.

pdfname= *file_path*
Specifies the directories and names for the PDF files and any existing PDF map files. By default, if the PDFDIR environment variable is set, the compiler places the PDF and PDF map files in the directory specified by PDFDIR. Otherwise, if the PDFDIR environment variable is not set, the compiler places these files in the current working directory. If the PDFDIR environment variable is set but the specified directory does not exist, the compiler issues a warning message. The name of the PDF map file follows the name of the PDF file if the

-qpdf1=unique option is not specified. For example, if you specify the **-qpdf1=pdfname=/home/joe/func** option, the generated PDF file is called func, and the PDF map file is called func_map. Both of the files are placed in the /home/joe directory. You can use the **pdfname** suboption to do simultaneous runs of multiple executable applications using the same directory. This is especially useful when you are tuning dynamic libraries with PDF.

unique | nounique

You can use the **-qpdf1=unique** option to avoid locking a single PDF file when multiple processes are writing to the same PDF file in the PDF training step. This option specifies whether a unique PDF file is created for each process during run time. The PDF file name is *<pdf_file_name>.<pid>*. *<pdf_file_name>* is ._pdf by default or specified by other **-qpdf1** suboptions, which include **pdfname**, **exename**, and **defname**. *<pid>* is the ID of the running process in the PDF training step. For example, if you specify the **-qpdf1=unique:pdfname=abc** option, and there are two processes for PDF training with the IDs 12345678 and 87654321, two PDF files abc.12345678 and abc.87654321 are generated.

Note: When **-qpdf1=unique** is specified, multiple PDF files with process IDs as suffixes are generated. You must use the **mergepdf** program to merge all these PDF files into one after the PDF training step.

Usage

The PDF process consists of the following three steps:

1. Compile your program with the **-qpdf1** option and a minimum optimization level of **-O2**. By default, a PDF map file named ._pdf_map and a resulting application are generated.
2. Run the resulting application with a typical data set. Profiling information is written to a PDF file named ._pdf by default. This step is called the PDF training step.
3. Recompile and link or just relink the program with the **-qpdf2** option and the optimization level used with the **-qpdf1** option. The **-qpdf2** process fine-tunes the optimizations according to the profiling information collected when the resulting application is run.

Notes:

- The **showpdf** utility uses the PDF map file to display part of the profiling information in text or XML format. For details, see "Viewing profiling information with showpdf" in the *XL C/C++ Optimization and Programming Guide*. If you do not need to view the profiling information, specify the **-qnoshowpdf** option during the **-qpdf1** phase so that the PDF map file is not generated. For details of **-qnoshowpdf**, see **-qshowpdf** in the *XL C/C++ Compiler Reference*.
- When any level of option **-qipa** is in effect, and you specify the **-qpdf1** or **-qpdf2** option at the link step but not at the compile step, the compiler issues a warning message. The message indicates that you must recompile your program to get all the profiling information.
- When the **-qpdf1=pdfname** option is used during the **-qpdf1** phase, you must use the **-qpdf2=pdfname** option during the **-qpdf2** phase for the compiler to recognize the correct PDF file. This rule also applies to the **-qpdf[1|2]=exename** option.

The compiler issues an information message with a number in the range of 0 - 100 during the **-qpdf2** phase. If you have not changed your program between the

-qpdf1 and **-qpdf2** phases, the number is 100, which means that all the profiling information can be used to optimize the program. If the number is 0, it means that the profiling information is completely outdated, and the compiler cannot take advantage of any information. When the number is less than 100, you can choose to recompile your program with the **-qpdf1** option and regenerate the profiling information.

If you recompile your program by using the **-qpdf1** option with any suboption, the compiler removes the existing PDF file or files whose names and locations are the same as the file or files that will be created in the training step before generating a new application.

Other related options

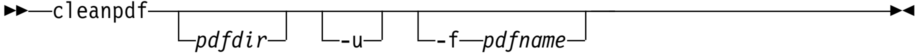
You can use the following option with the **-qpdf1** option:

-qshowpdf
: Uses the **showpdf** utility to view the PDF data that were collected. See "-qshowpdf" on page 134 for more information.

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

The following utility programs, found in `/opt/ibm/xlC/1.2.0/bin/`, are available for managing the files to which profiling information is written:

cleanpdf



Removes all PDF files or the specified PDF files, including PDF files with process ID suffixes. Removing profiling information reduces runtime overhead if you change the program and then go through the PDF process again.

pdfdir
: Specifies the directory that contains the PDF files to be removed. If *pdfdir* is not specified, the directory is set by the PDFDIR environment variable; if PDFDIR is not set, the directory is the current directory.

-f *pdfname*
: Specifies the name of the PDF file to be removed. If `-f` *`pdfname`* is not specified, `._pdf` is removed.

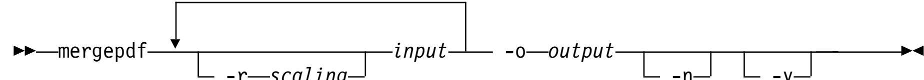
-u
: If `-f` *`pdfname`* is specified, in addition to the file removed by **-f**, files with the naming convention *`pdfname`*`.`<*`pid`*>, if applicable, are also removed.

  If `-f` *`pdfname`* is not specified, removes `._pdf`. Files with the naming convention `._pdf.`<*`pid`*>, if applicable, are also removed.

  <*pid*> is the ID of the running process in the PDF training step.

Run **cleanpdf** only when you finish the PDF process for a particular application. Otherwise, if you want to resume by using PDF process with that application, you must compile all of the files again with **-qpdf1**.

mergepdf



Merges two or more PDF files into a single PDF file.

-r *scaling*
: Specifies the scaling ratio for the PDF file. This value must be greater than zero and can be either an integer or a floating-point value. If not specified, a ratio of 1.0 is assumed.

input
: Specifies the name of a PDF input file, or a directory that contains PDF files.

-o *output*
: Specifies the name of the PDF output file, or a directory to which the merged output is written.

-n
: Specifies that PDF files do not get normalized. By default, **mergepdf** normalizes the files in such a way that every profile has the same overall weighting, and individual counters are scaled accordingly. This is done before applying the user-specified ratio (with **-r**). When **-n** is specified, no normalization occurs. If neither **-n** nor **-r** is specified, the PDF files are not scaled at all.

-v
: Specifies verbose mode, and causes internal and user-specified scaling ratios to be displayed to standard output.

showpdf

Displays part of the profiling information written to PDF and PDF map files. To use this command, you must first compile your program with the **-qpdf1** option. See "Viewing profiling information with showpdf" in the *XL C/C++ Optimization and Programming Guide* for more information.

Predefined macros

None.

Examples

The following example uses the **-qpdf1=level=0** option to reduce possible runtime instrumentation overhead:

```
#Compile all the files with -qpdf1=level=0
xlc -qpdf1=level=0 -O3 file1.c file2.c file3.c

#Run with one set of input data
./a.out < sample.data

#Recompile all the files with -qpdf2
xlc -qpdf2 -O3 file1.c file2.c file3.c

#If the sample data is typical, the program
#can now run faster than without the PDF process
```

The following example uses the **-qpdf1=level=1** option:

```
#Compile all the files with -qpdf1
xlc -qpdf1 -O3 file1.c file2.c file3.c

#Run with one set of input data
./a.out < sample.data
```

```
#Recompile all the files with -qpdf2
xlc -qpdf2 -O3 file1.c file2.c file3.c

#If the sample data is typical, the program
#can now run faster than without the PDF process
```

The following example demonstrates the use of the **-qpdf[1|2]=exename** option:

```
#Compile all the files with -qpdf1=exename
xlc -qpdf1=exename -O3 -o final file1.c file2.c file3.c

#Run executable with sample input data
./final < typical.data

#List the content of the directory
 >ls -lrta

 -rw-r--r-- 1 user staff 50 Dec 05 13:18 file1.c
 -rw-r--r-- 1 user staff 50 Dec 05 13:18 file2.c
 -rw-r--r-- 1 user staff 50 Dec 05 13:18 file3.c
 -rwxr-xr-x 1 user staff 12243 Dec 05 17:00 final
 -rwxr-Sr-- 1 user staff 762 Dec 05 17:03 .final_pdf

#Recompile all the files with -qpdf2=exename
xlc -qpdf2=exename -O3 -o final file1.c file2.c file3.c

#The program is now optimized using PDF information
```

The following example demonstrates the use of the **-qpdf[1|2]=pdfname** option:

```
#Compile all the files with -qpdf1=pdfname. The static profiling
#information is recorded in a file named final_map
xlc -qpdf1=pdfname=final -O3 file1.c file2.c file3.c

#Run executable with sample input data. The profiling
#information is recorded in a file named final
./a.out < typical.data

#List the content of the directory
 >ls -lrta

 -rw-r--r-- 1 user staff 50 Dec 05 13:18 file1.c
 -rw-r--r-- 1 user staff 50 Dec 05 13:18 file2.c
 -rw-r--r-- 1 user staff 50 Dec 05 13:18 file3.c
 -rwxr-xr-x 1 user staff 12243 Dec 05 18:30 a.out
 -rwxr-Sr-- 1 user staff 762 Dec 05 18:32 final

#Recompile all the files with -qpdf2=pdfname
xlc -qpdf2=pdfname=final -O3 file1.c file2.c file3.c

#The program is now optimized using PDF information
```

Related information

- “-qshowpdf” on page 134
- “-qipa” on page 109
- “-qreport” on page 130
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*
- “Runtime environment variables” on page 18
- "Profile-directed feedback" in the *XL C/C++ Optimization and Programming Guide*

-qpriority (C++ only)

Category

Object code control

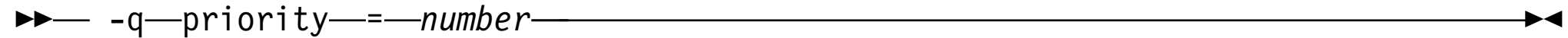
Purpose

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. You can use the **-qpriority** option to impose a construction order for all static objects declared within the same load module. Destructors for these objects are run in reverse order during termination.

Syntax

Option syntax



Defaults

The default priority level is 65535.

Parameters

number
An integer literal in the range of 101 to 65535. A lower value indicates a higher priority; a higher value indicates a lower priority. If you do not specify a *number*, the compiler assumes 65535.

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. See "The init_priority variable attribute" in the *XL C/C++ Language Reference* for more information.

Examples

To compile the file `myprogram.C` to produce an object file `myprogram.o` so that objects within that file have an initialization priority of 2000, enter the following command:

```
xlc++ myprogram.C -c -qpriority=2000
```

Related information

- "Initializing static objects in libraries" in the *XL C/C++ Optimization and Programming Guide*

-qreport

Category

Listings, messages, and compiler information

Pragma equivalent

None.

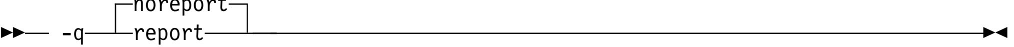
Purpose

Produces listing files that show how sections of code have been optimized.

A listing file is generated with a .lst suffix for each source file that is listed on the command line. When you specify **-qreport** with an option that enables vectorization, the listing file shows a pseudo-C code listing and a summary of how program loops are optimized. The report also includes diagnostic information about why specific loops cannot be vectorized. For example, when **-qreport** is specified with **-ftree-vectorize (-qsimd)**, messages are provided to identify non-stride-one references that prevent loop vectorization.

The compiler also reports the number of streams created for a given loop, which include both load and store streams. This information is included in the `Loop Transformation` section of the listing file. You can use this information to understand your application code and to tune your code for better performance. For example, you can distribute a loop which has more streams than the number supported by the underlying architecture.

Syntax



Defaults

-qnoreport

Usage

To generate a loop transformation listing, you must specify **-qreport** with one of the following options:
- **-qhot**
- **-O3** or higher

To generate PDF information in the listing, you must specify both **-qreport** and **-qpdf2**.

To generate data reorganization information, specify **-qreport** with the optimization level **-qipa=level=2**. Reorganizations include array splitting, array transposing, memory allocation merging, array interleaving, and array coalescing.

To generate information about data prefetch insertion locations, specify **-qreport** with the optimization level of **-qhot** or any other option that implies **-qhot**. This information appears in the `LOOP TRANSFORMATION SECTION` of the listing file.

The pseudo-C code listing is not intended to be compilable. Do not include any of the pseudo-C code in your program, and do not explicitly call any of the internal routines whose names may appear in the pseudo-C code listing.

Predefined macros

None.

Examples

To compile `myprogram.c` so the compiler listing includes a report showing how loops are optimized, enter:

```
xlc -qhot -O3 -qreport myprogram.c
```

Related information

- "-qhot" on page 104
- "-ftree-vectorize (-qsimd)" on page 78
- "-qipa" on page 109

-qrtti, -fno-rtti (-qnortti) (C++ only)

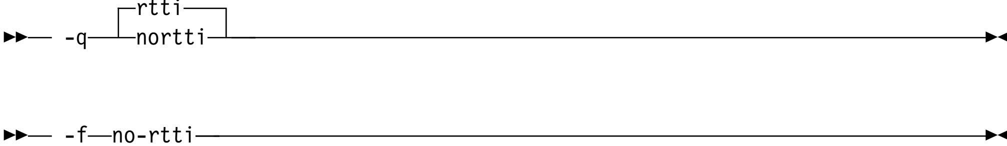
Category

Object code control

Purpose

Generates runtime type identification (RTTI) information for exception handling and for use by the `typeid` and `dynamic_cast` operators.

Syntax



Defaults

-fno-rtti (-qnortti)

Usage

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

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

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

Predefined macros

- __GXX_RTTI is predefined to a value of 1 when **-qrtti** is in effect; otherwise, it is undefined.
- __NO_RTTI__ is defined to 1 when **-fno-rtti (-qnortti)** is in effect; otherwise, it is undefined.
- __RTTI_ALL__ is defined to 1 when **-qrtti** is in effect; otherwise, it is undefined.
- __RTTI_DYNAMIC_CAST__ is predefined to a value of 1 when **-qrtti** is in effect; otherwise, it is undefined.
- __RTTI_TYPE_INFO__ is predefined to a value of 1 when **-qrtti** is in effect; otherwise, it is undefined.

Related information

- "-qeh (C++ only)" on page 101

-qsaveopt

Category

Object code control

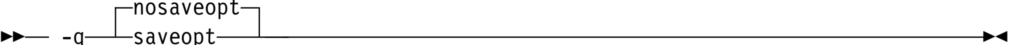
Pragma equivalent

None.

Purpose

Saves the command-line options used for compiling a source file, the user's configuration file name and the options specified in the configuration files, the version and level of each compiler component invoked during compilation, and other information to the corresponding object file.

Syntax



Defaults

-qnosaveopt

Usage

This option has effect only when compiling to an object (.o) file (that is, using the **-c** option). Though each object might contain multiple compilation units, only one copy of the command-line options is saved. Compiler options specified with pragma directives are ignored.

Command-line compiler options information is copied as a string into the object file, using the following format:

```
►►──@(#)──opt──┬─c─┬──invocation──options──────────────────────►◄
               └─C─┘
```

```
►►──@(#)──cfg────config_file_options_list──────────────────────────────►◄
```

```
►►──@(#)──env────env_var_definition────────────────────────────────────►◄
```

where:

c Signifies a C language compilation.

C Signifies a C++ language compilation.

invocation
Shows the command used for the compilation, for example, **xlc**.

options The list of command line options specified on the command line, with individual options separated by space.

config_file_options_list
The list of options specified by the **options** attribute in all configuration files that take effect in the compilation, separated by space.

env_var_definition
The environment variables that are used by the compiler. Currently only **XLC_USR_CONFIG** is listed.

Note: You can always use this option, but the corresponding information is only generated when the environment variable **XLC_USR_CONFIG** is set.

For more information about the environment variable **XLC_USR_CONFIG**, see Compile-time and link-time environment variables.

Note: The string of the command-line options is truncated after 64,000 bytes.

Compiler version and release information, as well as the version and level of each component invoked during compilation, are also saved to the object file in the format:

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

where:

V Represents the version.

R Represents the release.

M Represents the modification.

L Represents the level.

component_name
Specifies the components that were invoked for this compilation, such as the low-level optimizer.

product_name
Indicates the product to which the component belongs (for example, C/C++).

YYMMDD
Represents the year, month, and date of the installed update. If the update installed is at the base level, the level is displayed as BASE.

component_level_ID
Represents the ID associated with the level of the installed component.

If you want to simply output this information to standard output without writing it to the object file, use the **--version (-qversion)** option.

Predefined macros

None.

Examples

Compile `t.c` with the following command:

```
xlc t.c -c -qsaveopt -qhot
```

Issuing the **strings -a** command on the resulting `t.o` object file produces information similar to the following:

```
IBM XL C/C++ for Linux on z Systems, Version 1.2.0.0
@(#)opt c /opt/ibm/xlC/1.2.0/bin/xlc t.c -c -qsaveopt -qhot
@(#)cfg -qlanglvl=extc99 -qcpluscmt -qalias=ansi -ftls-model -D_CALL_SYSV
 -D__null=0 -D__NO_MATH_INLINES -ftls-model
@(#)version IBM XL C/C++ for Linux on z Systems, V1.2
@(#)version Version: 01.02.0000.0000
@(#)version Driver Version: 01.02(C/C++) Level: YYMMDD ID: _JbNFoYQ_EeWg_O7EssfHAg
@(#)version C/C++ Front End Version : 01.02(C/C++)  Level: YYMMDD ID: _JX7IIIQ_EeWg_O7EssfHAg
@(#)version High-Level Optimizer Version: 01.02(C/C++)  Level: YYMMDD
ID: _JfAAgYQ_EeWg_O7EssfHAg
@(#)version Low-Level Optimizer Version: 01.02(C/C++)  Level: YYMMDD
ID: _sk208X8mEeWg_O7EssfHAg
```

In the first line, `c` identifies the source used as C, `/opt/ibm/xlC/1.2.0/bin/xlc` shows the invocation command used, and `-qhot -qsaveopt` shows the compilation options.

The remaining lines list each compiler component invoked during compilation, and its version and level. Components that are shared by multiple products may show more than one version number. Level numbers shown may change depending on the updates you have installed on your system.

Related information

- "--version (-qversion)" on page 38

-qshowpdf

Category

Optimization and tuning

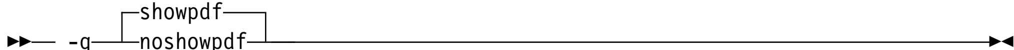
Pragma equivalent

None.

Purpose

When used with **-qpdf1** and a minimum optimization level of **-O2** at compile and link steps, creates a PDF map file that contains additional profiling information for all procedures in your application.

Syntax



Defaults

-qshowpdf

Usage

After you run your application with typical data, the profiling information is recorded into a profile-directed feedback (PDF) file (by default, the file is named `._pdf`).

In addition to the PDF file, the compiler also generates a PDF map file that contains static information during the **-qpdf1** phase. With these two files, you can use the **showpdf** utility to view part of the profiling information of your application in text or XML format. For details of the **showpdf** utility, see "Viewing profiling information with showpdf" in the *XL C/C++ Optimization and Programming Guide*.

If you do not need to view the profiling information, specify the **-qnoshowpdf** option during the **-qpdf1** phase so that the PDF map file is not generated. This can reduce your compile time.

Predefined macros

None.

Related information

- "-qpdf1, -qpdf2" on page 123
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*

-qsmallstack

Category

Optimization and tuning

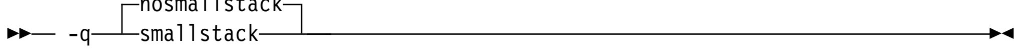
Pragma equivalent

None.

Purpose

Minimizes stack usage where possible. Disables optimizations that increase the size of the stack frame.

Syntax



Defaults

-qnosmallstack

Usage

Programs that allocate large amounts of data to the stack, such as threaded programs, might result in stack overflows. The **-qsmallstack** option helps avoid stack overflows by disabling optimizations that increase the size of the stack frame.

This option takes effect only when used together with IPA (the **-qipa** compiler option).

Specifying this option might adversely affect program performance.

Predefined macros

None.

Examples

To compile `myprogram.c` to use a small stack frame, enter the command:

```
xlc myprogram.c -qipa -qsmallstack
```

Related information

- "-g" on page 80
- "-qipa" on page 109
- "-O, -qoptimize" on page 50

-qstaticinline (C++ only)

Category

Language element control

Pragma equivalent

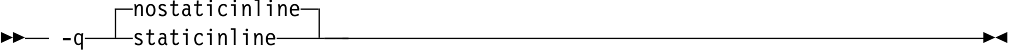
None.

Purpose

Controls whether inline functions are treated as having `static` or `extern` linkage.

When **-qnostaticinline** is in effect, the compiler treats inline functions as `extern`: only one function body is generated for a function marked with the `inline` function specifier, regardless of how many definitions of the same function appear in different source files. When **-qstaticinline** is in effect, the compiler treats inline functions as having `static` linkage: a separate function body is generated for each definition in a different source file of the same function marked with the `inline` function specifier.

Syntax



Defaults

-qnostaticinline

Usage

When **-qnostaticinline** is in effect, any redundant functions definitions for which no bodies are generated are discarded by default.

Predefined macros

None.

Examples

Using the **-qstaticinline** option causes function `f` in the following declaration to be treated as static, even though it is not explicitly declared as such. A separate function body is created for each definition of the function. Note that this can lead to a substantial increase in code size.

```
inline void f() {/*...*/};
```

-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)

Category

Input control

Purpose

Specifies whether the standard include directories are included in the search paths for system and user header files.

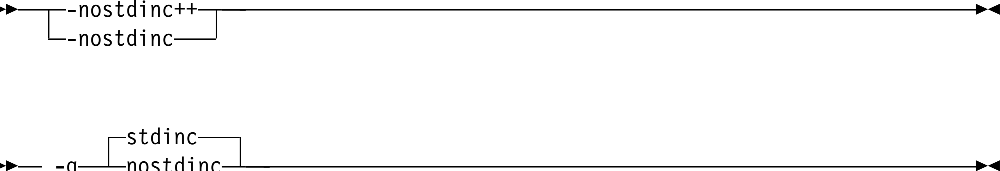
When **-qstdinc** is in effect, the compiler searches the following directories for header files:

- C The directory specified in the configuration file for the XL C header files (this is normally /opt/ibm/xlC/1.2.0/include/) or by the **-isystem (-qc_stdinc)** option
- C++ The directory specified in the configuration file for the XL C and C++ header files (this is normally /opt/ibm/xlC/1.2.0/include/) or by the **-isystem (-qcpp_stdinc)** option
- The directory specified in the configuration file for the system header files or by the **-isystem** (**-qgcc_c_stdinc** or **-qgcc_cpp_stdinc**) option

When **-nostdinc++** or **-nostdinc** (**-qnostdinc**) is in effect, these directories are excluded from the search paths. The following directories are searched:

- Directories in which source files containing `#include "filename"` directives are located
- Directories specified by the **-I** option
- Directories specified by the **-include (-qinclude)** option

Syntax



Defaults

-qstdinc

Usage

The search order of header files is described in “Directory search sequence for included files” on page 9.

This option only affects search paths for header files included with a relative name; if a full (absolute) path name is specified, this option has no effect on that path name.

The last valid pragma directive remains in effect until replaced by a subsequent pragma.

Predefined macros

None.

Examples

To compile `myprogram.c` so that *only* the directory `/tmp/myfiles` (in addition to the directory containing `myprogram.c`) is searched for the file included with the `#include "myinc.h"` directive, enter:

```
xlc myprogram.c -nostdinc -I/tmp/myfiles
```

Related information

- “-isystem (-qc_stdinc) (C only)” on page 85
- “-isystem (-qcpp_stdinc) (C++ only)” on page 87
- “-isystem (-qgcc_c_stdinc) (C only)” on page 88
- “-isystem (-qgcc_cpp_stdinc) (C++ only)” on page 89
- “-I” on page 48
- “Directory search sequence for included files” on page 9

-qtimestamps

Category

“Output control” on page 23

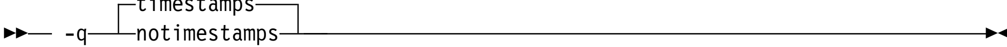
Pragma equivalent

None.

Purpose

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

Syntax



Defaults

-qtimestamps

Usage

By default, the compiler inserts an implicit time stamp in an object file when it is created. In some cases, comparison tools may not process the information in such binaries properly. Controlling time stamp generation provides a way of avoiding such problems. To omit the time stamp, use the option **-qnotimestamps**.

This option does not affect time stamps inserted by pragmas and other explicit mechanisms.

-qtmplinst (C++ only)

Category

Template control

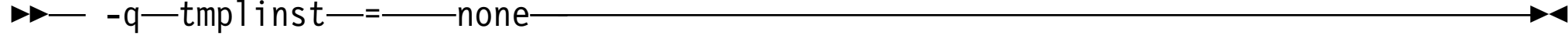
Pragma equivalent

None.

Purpose

Manages the implicit instantiation of templates.

Syntax



Defaults

-qtmplinst=none

Parameters

none
Instructs the compiler to instantiate only inline functions. No other implicit instantiation is performed.

Predefined macros

None.

Related information

- "Explicit instantiation" in the *XL C/C++ Optimization and Programming Guide*

-r

Category

Object code control

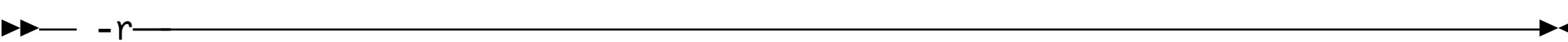
Pragma equivalent

None.

Purpose

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

Syntax



Defaults

Not applicable.

Usage

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

Predefined macros

None.

Examples

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

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

-s

Category

Object code control

Pragma equivalent

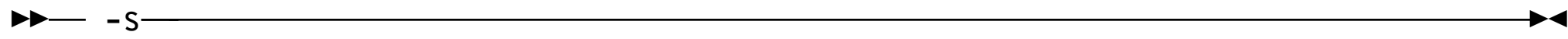
None.

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



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 80

-shared (-qmkshrobj)

Category

Output control

Pragma equivalent

None.

Purpose

Creates a shared object from generated object files.

Use this option, together with the related options described later in this topic, instead of calling the linker directly to create a shared object. The advantages of using this option are the automatic handling of link-time C++ template instantiation (using either the template include directory or the template registry), and compatibility with **-qipa** link-time optimizations.

Syntax

Defaults

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

Usage

The compiler automatically exports all global symbols from the shared object unless you specify which symbols to export by using the **--version-script** linker option. IBM Symbols that have the hidden or internal visibility attribute are not exported. IBM

Specifying **-shared (-qmkshrobj)** implies **-fPIC (-qpic)**.

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

-o *shared_file*
: The name of the file that holds the shared file information. The default is a.out.

-e *name*
: Sets the entry name for the shared executable to *name*.

-qstaticlink=xllibs
: When you specify **-qstaticlink=xllibs** and **-qmkshrobj**, both options take effect. The compiler creates a shared object in which all references to the XL libraries are statically linked in.

For detailed information about using **-shared (-qmkshrobj)** to create shared libraries, see "Constructing a library" in the *XL C/C++ Optimization and Programming Guide*.

Predefined macros

None.

Examples

To construct the shared library `big_lib.so` from three smaller object files, enter the following command:

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

Related information

- "-e" on page 60
- "-qipa" on page 109
- "-o" on page 97
- "-fPIC , -fpic (-qpic)" on page 69
- "-qpriority (C++ only)" on page 129
- "Supported GCC pragmas" on page 160
- "-static (-qstaticlink)"

-static (-qstaticlink)

Category

Linking

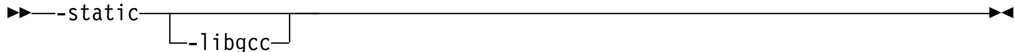
Pragma equivalent

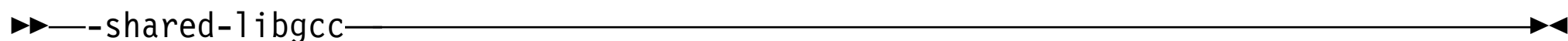
None.

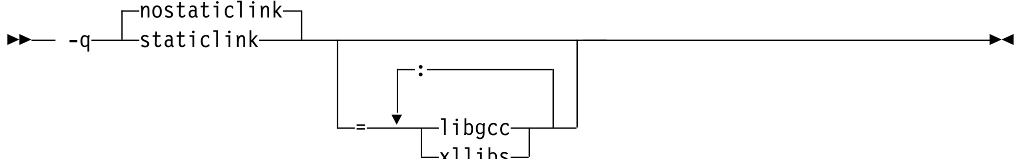
Purpose

Controls whether static or shared runtime libraries are linked into an application.

Syntax







The following table shows the equivalent usage between different format of options for specifying the linkage of shared and nonshared libraries.

Table 22. Option equivalence mapping

Equivalent option	Meaning
-static or -qstaticlink	Build a static object and prevent linking with shared libraries. Every library that is linked to must be a static library.
-shared-libgcc or -qnostaticlink=libgcc	Link with the shared version of libgcc.
-static-libgcc or -qstaticlink=libgcc	Link with the static version of libgcc.

Defaults

-qnostaticlink

Parameters

`libgcc`

- When you specify **-shared-libgcc**, the compiler links the shared version of **`libgcc`**.
- When you specify **-static-libgcc**, the compiler links the static version of **`libgcc`**.

`xllibs`

- When you specify **`xllibs`** with **-qnostaticlink**, the compiler links the shared version of the XL compiler libraries.
- When you specify **`xllibs`** with **-qstaticlink**, the compiler links the static version of the XL compiler libraries.

The **xllibs** suboption is available only for the **-qstaticlink** and **-qnostaticlink** options.

Usage

When you specify **-static** without suboptions, only static libraries are linked with the object file.

When you specify **-qnostaticlink** without suboptions, shared libraries are linked with the object file.

When you specify **-qstaticlink=xllibs** and **-qmkshrobj**, both options take effect. The compiler links in the static version of XL libraries and creates a shared object at the same time.

When compiler options are combined, conflicts might occur. The following table describes the resolutions of the conflicting compiler options.

Table 23. Examples of conflicting compiler options and resolutions

<table>
<tr><th>Options combination examples</th><th>Resolution result</th><th>Compiler behavior</th></tr>
<tr><td>-qnostaticlink -static-libgcc</td><td>Equivalent to -static-libgcc</td><td rowspan="2">If you first specify -qnostaticlink without suboptions and then specify -static or -qstaticlink with or without suboptions, -qnostaticlink is overridden. All libraries are linked statically.</td></tr>
<tr><td>-qnostaticlink -qstaticlink=xllibs</td><td>Equivalent to -qstaticlink=xllibs</td></tr>
<tr><td>-static-libgcc -qnostaticlink</td><td>Equivalent to -qnostaticlink</td><td>If you specify -static with or without suboptions followed by -qnostaticlink without suboptions, -qnostaticlink takes effect and shared libraries are linked.</td></tr>
<tr><td>-static -shared-libgcc</td><td>Equivalent to -static</td><td rowspan="2">If you specify -static without suboptions followed by -shared-libgcc or -qnostaticlink with suboptions, -static takes effect and only static libraries are linked with the object file.</td></tr>
<tr><td>-static -qnostaticlink=libgcc:xllibs</td><td>Equivalent to -static</td></tr>
<tr><td>-shared-libgcc -static</td><td>Equivalent to -static</td><td>If you first specify -shared-libgcc with suboptions and then specify -static without suboptions, -static takes effect and all libraries are linked statically.</td></tr>
</table>

Notes:

- If a runtime library is linked in statically while its message catalog is not installed on the system, messages are issued with message numbers only, and no message text is shown.
- If a shared library or a dynamically linked application is supposed to throw or catch exceptions, you must link it with the shared **libgcc** by using **-shared-libgcc**.

Predefined macros

None.

Related information

- "-shared (-qmkshrobj)" on page 141

-std (-qlanglvl)

Category

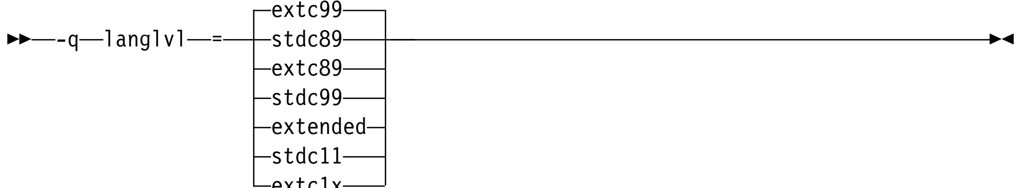
Language element control

Purpose

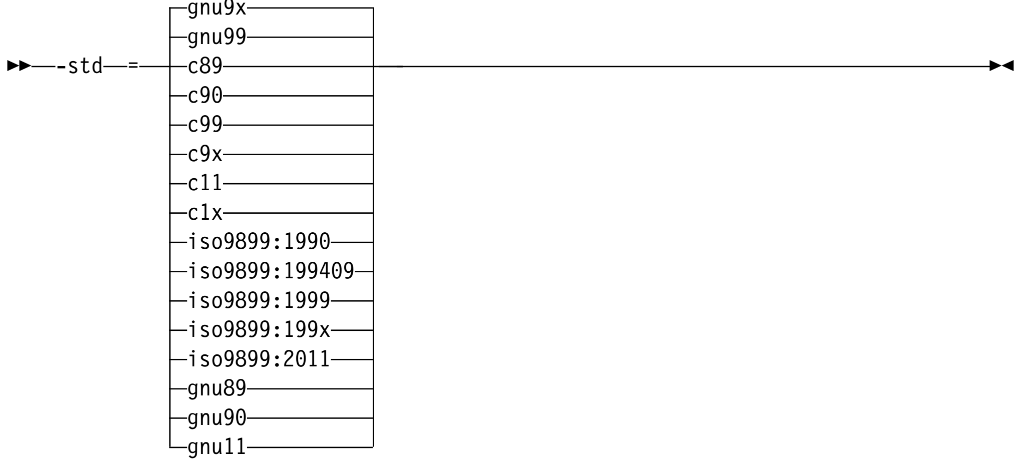
Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.

Syntax

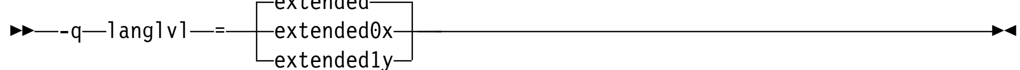
-qlanglvl syntax (C only)



-std syntax (C only)



-qlanglvl syntax (C++ only)



-std syntax (C++ only)

Defaults

- C **-std=gnu99** or **-std=gnu9x**
- C++ **-std=gnu++98**
- C The default is set according to the command used to invoke the compiler:

- **-qlanglvl=extc99** for the **xlc** and related invocation commands
- **-qlanglvl=extended** for the **cc** and related invocation commands
- **-qlanglvl=stdc89** for the **c89** and related invocation commands
- **-qlanglvl=stdc99** for the **c99** and related invocation commands

- C++ The default is set according to the command used to invoke the compiler:
  - **-qlanglvl=extended** for the **xlC** or **xlc++** and related invocation commands

Parameters for C language programs

Parameters of the -std option:

c89 | c90 | iso9899:1990
: Compilation conforms strictly to the ANSI C89 standard, also known as ISO C90.

iso9899:199409
: Compilation conforms strictly to the ISO C95 standard.

c99 | c9x | iso9899:1999 | iso9899:199x
: Compilation conforms strictly to the ISO C99 standard, also known as ISO C99.

C11 **c11**[1] **| c1x | iso9899:2011**
: Compilation conforms strictly to the ISO C11 standard. C11

gnu89 | gnu90
: Compilation conforms to the ANSI C89 standard and accepts implementation-specific language extensions, also known as GNU C90.

gnu99 | gnu9x
: Compilation conforms to the ISO C99 standard and accepts implementation-specific language extensions, also known as GNU C99.

gnu11[1]
: Compilation conforms to the ISO C11 standard and accepts implementation-specific language extensions, also known as GNU C11.

If you are using some of the C11 features, you must use the **-qlanglvl** option.

Parameters of the -qlanglvl option:

stdc89
: Compilation conforms strictly to the ANSI C89 standard, also known as ISO C90.

extc89
: Compilation conforms to the ANSI C89 standard and accepts implementation-specific language extensions.

stdc99
: Compilation conforms strictly to the ISO C99 standard.

extc99
: Compilation conforms to the ISO C99 standard and accepts implementation-specific language extensions.

extended
: Compilation is based on the ISO C89 standard, with some differences to accommodate extended language features.

C11 **stdc11**
Compilation conforms strictly to the ISO C11 standard. C11

C11 **extc1x**
Compilation is based on the C11 standard, invoking all the currently supported C11 features and other implementation-specific language extensions. C11

The following tables reflect the mapping between the **-qlanglvl** and **-std** suboptions:

Table 24. Mapping between the ***-qlanglvl*** *and* ***-std*** *suboptions (C only)*

-qlanglvl suboption	Mapping to -std suboption
stdc89	c89 | c90 | iso9899:1990
extc89	gnu89 | gnu90
stdc99	c99 | c9x | iso9899:1999 | iso9899:199x
extc99	gnu99 | gnu9x
stdc11	c11 | c1x | iso9899:2011
extc1x	gnu11

Parameters for C++ language programs

Parameters of the -std option:

gnu++98 | gnu++03
Compilation is based on the ISO C++98 standard, with some differences to accommodate extended language features.

c++98 | c++03
Compilation conforms strictly to the ISO C++ standard, also known as ISO C++98.

C++11 **c++11[1] | c++0x**
Compilation conforms strictly to the ISO C++ standard plus amendments, also known as ISO C++11.

Note: IBM supports the majority of C++11 features and will continue to develop and implement the features of this standard.
C++11

C++11 **gnu++11[1] | gnu++0x**
Compilation is based on the ISO C++ standard, with some differences to accommodate extended language features. C++11

C++14 **c++1y**
Compilation is based on the C++14 standard, invoking most of the C++11 features and all the currently supported C++14 features. C++14

Parameters of the -qlanglvl option:

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

C++11 **extended0x**
Compilation is based on the C++11 standard, invoking most of the C++ features and all the currently-supported C++11 features.

Note: IBM supports the majority of C++11 features and will continue to develop and implement the features of this standard.
C++11

C++14 **extended1y**
Compilation is based on the C++14 standard, invoking most of the C++11 features and all the currently supported C++14 features.

Note: IBM supports selected features of C++14 standard. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++14 features is complete, including the support of a new C++14 standard library, the implementation might change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++14 features.
C++14

The following tables reflect the mapping between the **-qlanglvl** and **-std** suboptions:

Table 25. Mapping between the **-qlanglvl** *and* **-std** *suboptions (C++ only)*

-qlanglvl suboption	Mapping to -std suboption
extended	**gnu++98** | **gnu++03**
extended0x	**gnu++11** | **gnu++0x**
extended1y	**c++1y**

Note:
1. This suboption is available only on platforms where GCC supports it.

Predefined macros

See "Macros related to language levels" on page 176 for a list of macros that are predefined by **-qlanglvl** suboptions.

-t

Category

Compiler customization

Pragma equivalent

None.

Purpose

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

Syntax

Defaults

The default paths for all of the compiler components are defined in the compiler configuration file.

Parameters

The following table shows the correspondence between **-t** parameters and the component names:

Parameter	Description	Component name
a	The assembler	as
b	The low-level optimizer	xlCcode
c, C	The C and C++ compiler front end	xlCentry
d	The disassembler	dis
I (uppercase i)	The high-level optimizer, compile step	ipa
L	The high-level optimizer, link step	ipa
l (lowercase L)	The linker	ld
p	The preprocessor	xlCentry

Usage

Use this option with the **-B***prefix* option. If **-B** is specified without the *prefix*, the default prefix is /lib/o. If **-B** is not specified at all, the prefix of the standard program names is /lib/n.

Note: If you use the **p** suboption, it can cause the source code to be preprocessed separately before compilation, which can change the way a program is compiled.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the name `/u/newones/compilers/` is prefixed to the compiler and assembler program names, enter:

```
xlc myprogram.c -B/u/newones/compilers/ -tca
```

Related information

- "-B" on page 42

-v, -V

Category

Listings, messages, and compiler information

Pragma equivalent

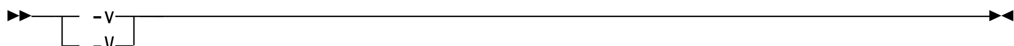
None.

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:

```
xlc myprogram.c -v
```

Related information

- "-### (-#) (pound sign)" on page 36

-w

Category

Listings, messages, and compiler information

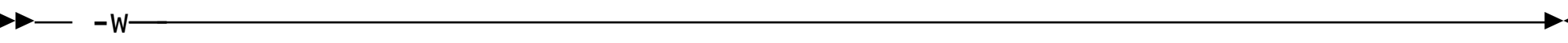
Pragma equivalent

None.

Purpose

Suppresses warning messages.

Syntax



Defaults

All informational and warning messages are reported.

Usage

Informational and warning messages that supply additional information to a severe error are not disabled by this option.

Predefined macros

None.

Examples

Consider the file `myprogram.c`.

```
#include <stdio.h>
int main()
 { char* greeting = "hello world";
   printf("%d \n", greeting);
   return 0;
}
```

- If you compile `myprogram.c` without the `-w` option, the compiler issues a warning message.

  ```
  xlC myprogram.c
  ```

  Output:

  ```
  "5:18: warning: format specifies type 'int' but the argument has type 'char *' [-Wformat]
  printf("%d \n", greeting);
  ~~ ^~~~~~
  %s
  1 warning generated."
  ```

- If you compile `myprogram.c` with the `-w` option, the warning message is suppressed.

  ```
  xlC myprogram.c -w
  ```

-x (-qsourcetype)

Category

Input control

Pragma equivalent

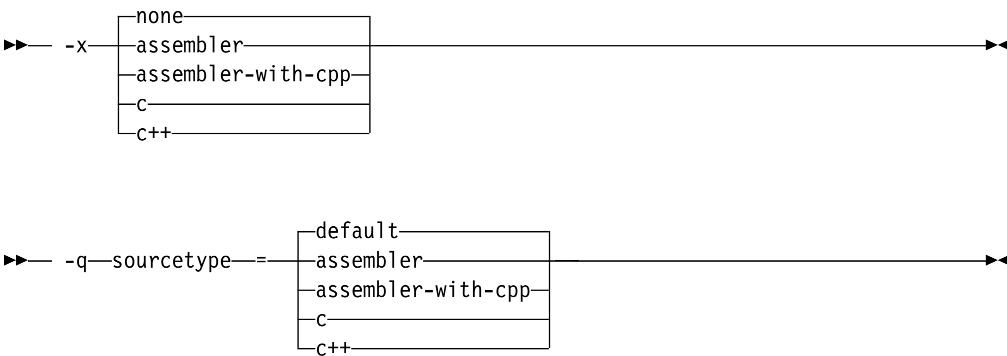
None.

Purpose

Instructs the compiler to treat all recognized source files as a specified source type, regardless of the actual file name suffix.

Ordinarily, the compiler uses the file name suffix of source files specified on the command line to determine the type of the source file. For example, a .c suffix normally implies C source code, and a .C suffix normally implies C++ source code. The **-x** option instructs the compiler to not rely on the file name suffix, and to instead assume a source type as specified by the option.

Syntax



Defaults

-x none or **-qsourcetype=default**

Parameters

assembler
: All source files following the option are compiled as if they are assembler language source files.

assembler-with-cpp
: All source files following the option are compiled as if they are assembler language source files that need preprocessing.

c
: All source files following the option are compiled as if they are C language source files.

c++
: All source files following the option are compiled as if they are C++ language source files. This suboption is equivalent to the **-+** option.

default (-qsourcetype only)
: The programming language of a source file is implied by its file name suffix.

none (-x only)
: The programming language of a source file is implied by its file name suffix.

Usage

If you do not use this option, files must have a suffix of .c to be compiled as C files, and .C (uppercase C), .cc, .cp, .cpp, .cxx, or .c++ to be compiled as C++ files.

Note that the option only affects files that are specified on the command line *following* the option, but not those that precede the option. Therefore, in the following example:

```
xlc goodbye.C -x c hello.C
```

`hello.C` is compiled as a C source file, but `goodbye.C` is compiled as a C++ file.

Predefined macros

None.

Related information

- "-+ (plus sign) (C++ only)" on page 37

Supported GCC options

The following GCC options are also supported in IBM XL C/C++ for Linux on z Systems, V1.2. For details about these options, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

- @file
- -###
- --help
- --sysroot
- --version
- -ansi
- -dD
- -dM
- -fansi-escape-codes
- -fasm, -fno-asm
- -fcolor-diagnostics
- -fcommon, -fno-common
- -fconstexpr-depth
- -fconstexpr-steps
- -fdiagnostic-parsable-fixits
- -fdiagnostic-show-category=[none|id|name]
- -fdiagnostic-show-template-tree
- -fdiagnostics-fixit-info
- -fdiagnostics-format=[clang|msvc|vi]
- -fdiagnostics-print-source-range-info
- -fdiagnostics-show-name
- -fdiagnostics-show-option
- -fdollars-in-identifiers, -fno-dollars-in-identifiers
- -fdump-class-hierarchy
- -fexceptions, -fno-exceptions

- -fexec-charset
- -ffreestanding
- -fgnu89-inline
- -fhosted
- -finline-functions
- -fmessage-length
- -fno-access-control
- -fno-assume-sane-operator-new
- -fno-builtin
- -fno-diagnostics-show-caret
- -fno-diagnostics-show-option
- -fno-elide-type
- -fno-gnu-keywords
- -fno-operator-names
- -fno-rtti
- -fno-show-column
- -fpack-struct
- -fpermissive
- -fpic, -fno-pic
- -fPIC, -fno-PIC
- -fPIE, -fno-PIE
- -fpie, -fno-pie
- -fshort-enums
- -fshort-wchar
- -fshow-column
- -fshow-source-location
- -fsigned-bitfields, -fno-signed-bitfields
- -fsigned-char, -fno-signed-char
- -fstrict-aliasing
- -fsyntax-only
- -ftabstop=*width*
- -ftemplate-backtrace-limit
- -ftemplate-depth
- -ftime-report
- -ftls-model, -fno-tls-model
- -ftrap-function=*name*
- -ftrapping-math, -fnotrapping-math
- -funsigned-bitfields, -fno-unsigned-bitfields
- -funsigned-char, -fno-unsigned-char
- -idirafter
- -imacros
- -include
- -iprefix
- -iquote
- -isysroot

- -isystem
- -iwithprefix
- -march
- -mtpf-trace
- -mtune
- -M
- -MD
- -MF
- -MG
- -MM
- -MMD
- -MP
- -MQ
- -MT
- -nodefaultlibs
- -nostartfiles
- -nostdinc
- -nostdinc++
- -pedantic
- -pedantic-errors
- -pie
- -rdynamic
- -shared
- -shared-libgcc
- -static
- -static-libgcc
- -std
- -trigraphs
- -w
- -Wall
- -Wambiguous-member-template
- -Wbad-function-cast
- -Wbind-to-temporary-copy
- -Wc++11-compat
- -Wcast-align
- -Wchar-subscripts
- -Wcomment
- -Wconversion
- -Wdelete-non-virtual-dtor
- -Wempty-body
- -Wenum-compare
- -Werror
- -Werror=foo [specically, -Werror=unused-command-line-argument to switch between warning/error for invalid options]
- -Weverything

- -Wextra-tokens
- -Wfatal-errors
- -Wfloat-equal
- -Wfoo
- -Wformat-nonliteral
- -Wformat-security
- -Wformat-y2k
- -Wignored-qualifiers
- -Wimplicit
- -Wimplicit-function-declaration
- -Wimplicit-int
- -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-extra-tokens
- -Wno-format
- -Wno-format-extra-args
- -Wno-format-zero-length
- -Wno-int-conversion
- -Wno-int-to-pointer-cast
- -Wno-invalid-offsetof
- -Wno-multichar
- -Wnonnull
- -Wno-return-local-addr
- -Wno-unused-result
- -Wno-virtual-move-assign
- -Wnon-virtual-dtor
- -Woverlength-strings
- -Woverloaded-virtual
- -Wpadded
- -Wparantheses
- -Wpedantic
- -Wpointer-arith
- -Wpointer-sign
- -Wreorder
- -Wreturn-type
- -Wsequence-point

- -Wshadow
- -Wsign-compare
- -Wsign-conversion
- -Wsizeof-pointer-memaccess
- -Wswitch
- -Wsystem-headers
- -Wtautological-compare
- -Wtrigraphs
- -Wtype-limits
- -Wundef
- -Wuninitialized
- -Wunknown-pragmas
- -Wunused
- -Wunused-label
- -Wunused-parameter
- -Wunused-value
- -Wunused-variable
- -Wvarargs
- -Wvariadic-macros
- -Wvla
- -Wwrite-strings
- -x
- -X

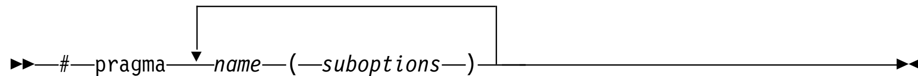
Chapter 4. Compiler pragmas reference

The following sections describe the available pragmas:
- "Pragma directive syntax"
- "Scope of pragma directives"
- "Supported GCC pragmas" on page 160
- "Supported IBM pragmas" on page 160

Pragma directive syntax

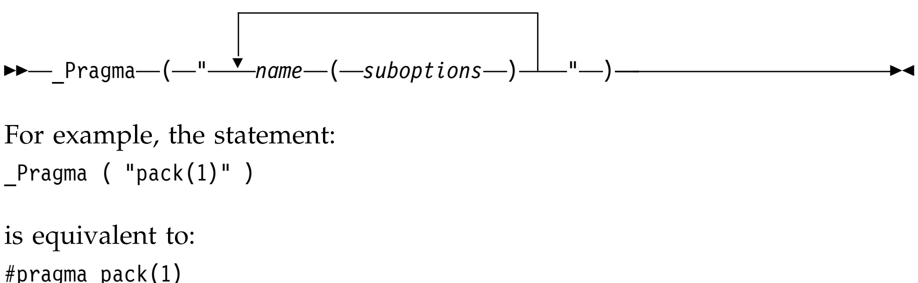
XL C/C++ supports the following forms of pragma directives:

#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

© Copyright IBM Corp. 2015

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 GCC pragmas

The following GCC pragmas are supported in IBM XL C/C++ for Linux on z Systems, V1.2. For details about these pragmas, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

- #pragma GCC dependency
- #pragma GCC diagnostic *kind option*
- #pragma GCC diagnostic pop
- #pragma GCC diagnostic push
- #pragma GCC error *string*
- #pragma GCC poison
- #pragma GCC system_header
- #pragma GCC visibility push(visibility)
- #pragma GCC visibility pop
- #pragma GCC warning *string*
- #pragma message *string*
- #pragma once
- #pragma pop_macro("*macro_name*")
- #pragma push_macro("*macro_name*")
- #pragma redefine_extname *oldname newname*
- #pragma unused

Supported IBM pragmas

This section contains descriptions of individual pragmas available in XL C/C++.

For each pragma, the following information is given:

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

Purpose
This section provides a brief description of the effect of the pragma, and why you might want to use it.

Syntax
This section provides the syntax for the pragma. For convenience, the **#pragma *name*** form of the directive is used in each case. However, it is perfectly valid to use the alternate C99-style `_Pragma` operator syntax; see “Pragma directive syntax” on page 159 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.

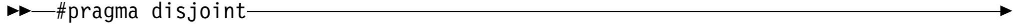
#pragma disjoint

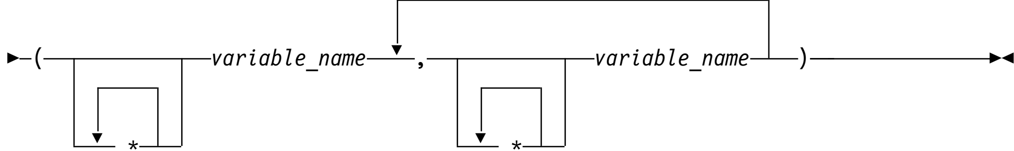
Purpose

Lists identifiers that are not aliased to each other within the scope of their use.

By informing the compiler that none of the identifiers listed in the pragma shares the same physical storage, the pragma provides more opportunity for optimizations.

Syntax





Parameters

variable_name
: The name of a variable. It must not refer to any of the following:
  - A member of a structure, class, or union
  - A structure, union, or enumeration tag
  - An enumeration constant
  - A `typedef` name
  - A label

Usage

The **#pragma disjoint** directive asserts that none of the identifiers listed in the pragma share physical storage; if any the identifiers *do* actually share physical storage, the pragma may give incorrect results.

The pragma can appear only in the function or block scope. An identifier in the directive must be visible at the point in the program where the pragma appears.

You must declare the identifiers before using them in the pragma. Your program must not dereference a pointer in the identifier list nor use it as a function

argument before it appears in the directive.

Examples

The following example shows the use of **#pragma disjoint**.

```
int a, b, *ptr_a, *ptr_b;

one_function()
{
  #pragma disjoint(*ptr_a, b)   /* *ptr_a never points to b */
  #pragma disjoint(*ptr_b, a)   /* *ptr_b never points to a */

  b = 6;
  *ptr_a = 7;   /* Assignment will not change the value of b  */

  another_function(b);    /* Argument "b" has the value 6  */
}
```

External pointer `ptr_a` does not share storage with and never points to the external variable `b`. Consequently, assigning 7 to the object to which `ptr_a` points will not change the value of `b`. Likewise, external pointer `ptr_b` does not share storage with and never points to the external variable `a`. The compiler can assume that the argument to `another_function` has the value 6 and will not reload the variable from memory.

#pragma 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--(--+-very_low--+--)-------------------><
                                       '-very_high-'
```

Parameters

very_low
: Marks source code that you expect will be executed very infrequently.

very_high
: Marks source code that you expect will be executed very frequently.

Usage

Use this pragma in conjunction with an optimization option; if optimization is not enabled, the pragma has no effect.

The pragma must be placed within block scope, and acts on the closest 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 nosimd

Purpose

Disables automatic generation of vector instructions. This pragma needs to be specified on a per-loop basis.

Syntax

```
►►──#──pragma──nosimd──────────────────────────────────────────►◄
```

Example

In the following example, `#pragma nosimd` is used to disable **-qsimd=auto** for a specific `for` loop.

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

Related reference:

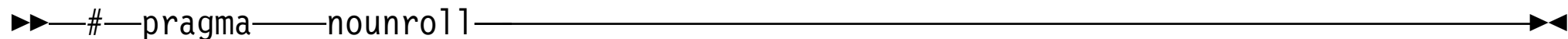
"-ftree-vectorize (-qsimd)" on page 78

#pragma nounroll

Purpose

Controls the loop not unrolling, for performance improvement.

Pragma syntax



Usage

Only one pragma can be specified for a given loop. The pragma must appear immediately before the loop to take effect. The pragma affects only the loop that follows it.

The **#pragma nounroll** directives can be used on `for` loops only. The **#pragma nounroll** directives cannot be applied to `do while` or `while` loops.

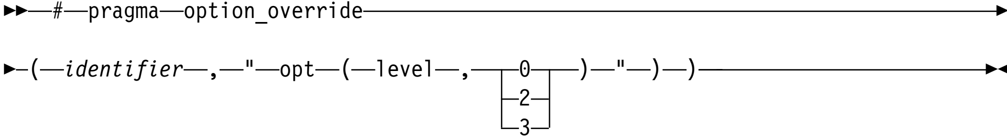
#pragma option_override

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



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[1]

#pragma option_override value	Equivalent compiler option
level, 2	-O2[1]
level, 3	-O3[2]

Notes:

1. If optimization level **-O3** is specified on the command line, `#pragma option_override(`*identifier*`, "opt(level, 0)")` or `#pragma option_override(`*identifier*`, "opt(level, 2)")` does not turn off the implication of the **-qhot** and **-qipa** options.
2. Specifying **-O3** implies **-qhot=level=0**. However, specifying `#pragma option_override(`*identifier*`, "opt(level, 3)")` in source code does not imply **-qhot=level=0**.

Defaults

See the descriptions for the options listed in the table above for default settings.

Usage

The pragma takes effect only if optimization is already enabled by a command-line option. You can only specify an optimization level in the pragma *lower* than the level applied to the rest of the program being compiled.

The **#pragma option_override** directive only affects functions that are defined in the same compilation unit. The pragma directive can appear anywhere in the translation unit. That is, it can appear before or after the function definition, before or after the function declaration, before or after the function has been referenced, and inside or outside the function definition.

C++ This pragma cannot be used with overloaded member functions.

Examples

Suppose you compile the following code fragment containing the functions `foo` and `faa` using **-O2**. Since it contains the `#pragma option_override(faa, "opt(level, 0)")`, function `faa` will not be optimized.

```
foo(){
     .
     .
     .
     }

#pragma option_override(faa, "opt(level, 0)")

faa(){
     .
     .
     .
     }
```

Related information

- “-O, -qoptimize” on page 50

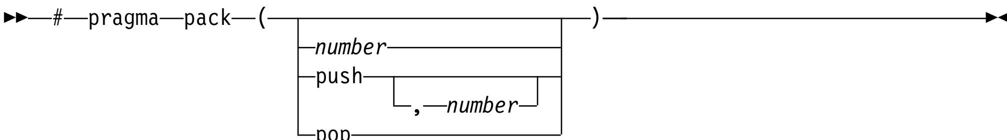
#pragma pack

Purpose

Sets the alignment of all aggregate members to a specified byte boundary.

If the byte boundary number is smaller than the natural alignment of a member, padding bytes are removed, thereby reducing the overall structure or union size.

Syntax



Defaults

Members of aggregates (structures, unions, and classes) are aligned on their natural boundaries and a structure ends on its natural boundary. The alignment of an aggregate is that of its strictest member (the member with the largest alignment requirement).

Parameters

number
: is one of the following:

1 Aligns structure members on 1-byte boundaries, or on their natural alignment boundary, whichever is less.

2 Aligns structure members on 2-byte boundaries, or on their natural alignment boundary, whichever is less.

4 Aligns structure members on 4-byte boundaries, or on their natural alignment boundary, whichever is less.

8 Aligns structure members on 8-byte boundaries, or on their natural alignment boundary, whichever is less.

16 Aligns structure members on 16-byte boundaries, or on their natural alignment boundary, whichever is less.

push
: When specified without a *number*, pushes whatever value is currently in effect to the top of the packing "stack". When used with a *number*, pushes that value to the top of the packing stack, and sets the packing value to that of *number* for structures that follow.

pop
: Removes the previous value added with **#pragma pack**. Specifying **#pragma pack()** with no parameters is equivalent to **#pragma pack(pop)**.

Usage

The **#pragma pack** directive applies to the definition of an aggregate type, rather than to the declaration of an instance of that type; it therefore automatically applies to all variables declared of the specified type.

The **#pragma pack** directive modifies the current alignment rule for only the members of structures whose declarations follow the directive. It does not affect the alignment of the structure directly, but by affecting the alignment of the members of the structure, it may affect the alignment of the overall structure.

The **#pragma pack** directive cannot increase the alignment of a member, but rather can decrease the alignment. For example, for a member with data type of `short`, a **#pragma pack(1)** directive would cause that member to be packed in the structure on a 1-byte boundary, while a **#pragma pack(4)** directive would have no effect.

The **#pragma pack** directive causes bit fields to cross bit field container boundaries.

```
#pragma pack(2)
struct A{
  int a:31;
  int b:2;
}x;

int main(){
  printf("size of struct A = %lu\n", sizeof(x));
}
```

When the program is compiled and run, the output is:

```
size of struct A = 6
```

But if you remove the **#pragma pack** directive, you get this output:

```
size of struct A = 8
```

The **#pragma pack** directive applies only to complete declarations of structures or unions; this excludes forward declarations, in which member lists are not specified. For example, in the following code fragment, the alignment for `struct S` is 4, since this is the rule in effect when the member list is declared:

```
#pragma pack(1)
struct S;
#pragma pack(4)
struct S { int i, j, k; };
```

A nested structure has the alignment that precedes its declaration, not the alignment of the structure in which it is contained, as shown in the following example:

```
#pragma pack (4)                // 4-byte alignment
         struct nested {
           int  x;
           char y;
           int  z;
         };

         #pragma pack(1)           // 1-byte alignment
         struct packedcxx{
            char   a;
            short  b;
            struct nested  s1;     // 4-byte alignment
         };
```

If more than one **#pragma pack** directive appears in a structure defined in an inlined function, the **#pragma pack** directive in effect at the beginning of the structure takes precedence.

Examples

The following example shows how the **#pragma pack** directive can be used to set the alignment of a structure definition:

```
//  header file file.h

   #pragma pack(1)

   struct jeff{           //    this structure is packed
     short bill;          //    along 1-byte boundaries
     int *chris;
   };
   #pragma pack(pop)       //   reset to previous alignment rule
// source file anyfile.c

   #include "file.h"

   struct jeff j;           //   uses the alignment specified
                            //   by the pragma pack directive
                            //   in the header file and is
                            //   packed along 1-byte boundaries
```

This example shows how a **#pragma pack** directive can affect the size and mapping of a structure:

```
struct s_t {
 char a;
 int b;
 short c;
 int d;
}S;
```

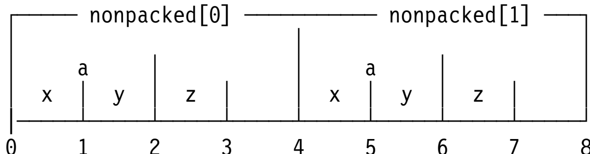
Default mapping:	**With #pragma pack(1):**
size of s_t = 16	size of s_t = 11
offset of a = 0	offset of a = 0
offset of b = 4	offset of b = 1
offset of c = 8	offset of c = 5
offset of d = 12	offset of d = 7
alignment of a = 1	alignment of a = 1
alignment of b = 4	alignment of b = 1
alignment of c = 2	alignment of c = 1
alignment of d = 4	alignment of d = 1

The following example defines a union uu containing a structure as one of its members, and declares an array of 2 unions of type uu:

```
union uu {
  short    a;
  struct {
    char x;
    char y;
    char z;
  } b;
};

union uu nonpacked[2];
```

Since the largest alignment requirement among the union members is that of `short a`, namely, 2 bytes, one byte of padding is added at the end of each union in the array to enforce this requirement:



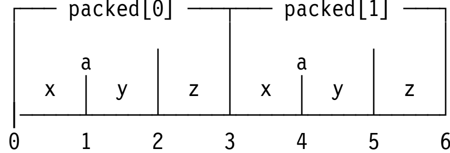
The next example uses **#pragma pack(1)** to set the alignment of unions of type `uu` to 1 byte:

```
#pragma pack(1)

 union uu {
   short    a;
   struct {
     char x;
     char y;
     char z;
   } b;
 };

 union uu pack_array[2];
```

Now, each union in the array `packed` has a length of only 3 bytes, as opposed to the 4 bytes of the previous case:



Related information

- “-fpack-struct (-qalign)” on page 70
- "Using alignment modifiers" in the *XL C/C++ Optimization and Programming Guide*

#pragma reachable

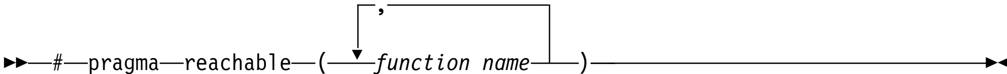
Purpose

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.

Note: The compiler automatically inserts **#pragma reachable** directives for the `setjmp` family of functions (`setjmp`, `_setjmp`, `sigsetjmp`, and `_sigsetjmp`) when you include the setjmp.h header file.

Syntax



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.

Chapter 5. Compiler predefined macros

Predefined macros can be used to conditionally compile code for specific compilers, specific versions of compilers, specific environments, and specific language features.

Predefined macros fall into several categories:
- "General macros"
- "Macros related to the platform" on page 173
- "Macros related to compiler features" on page 174

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 26. General predefined macros

Predefined macro name	Description	Predefined value
__BASE_FILE__	Indicates the name of the primary source file.	The fully qualified file name of the primary source file.
__DATE__	Indicates the date that the source file was preprocessed.	A character string containing the date when the source file was preprocessed.
__FILE__	Indicates the name of the preprocessed source file.	A character string containing the name of the preprocessed source file.
__FUNCTION__	Indicates the name of the function currently being compiled.	A character string containing the name of the function currently being compiled.
__LINE__	Indicates the current line number in the source file.	An integer constant containing the line number in the source file.
__SIZE_TYPE__	Indicates the underlying type of `size_t` on the current platform. Not protected.	`unsigned int` in 31-bit compilation mode and `unsigned long` in 64-bit compilation mode.
__TIME__	Indicates the time that the source file was preprocessed.	A character string containing the time when the source file was preprocessed.

© Copyright IBM Corp. 2015

Table 26. General predefined macros (continued)

Predefined macro name	Description	Predefined value
__TIMESTAMP__	Indicates the date and time when the source file was last modified. The value changes as the compiler processes any include files that are part of your source program.	A character string literal in the form "*Day Mmm dd hh:mm:ss yyyy*", where: *Day* Represents the day of the week (`Mon`, `Tue`, `Wed`, `Thu`, `Fri`, `Sat`, or `Sun`). *Mmm* Represents the month in an abbreviated form (`Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, or `Dec`). *dd* Represents the day. If the day is less than 10, the first `d` is a blank character. *hh* Represents the hour. *mm* Represents the minutes. *ss* Represents the seconds. *yyyy* Represents the year.

Macros indicating the XL C/C++ compiler

Macros related to the XL C/C++ compiler are always predefined, and they are protected, which means that the compiler will issue a warning if you try to undefine or redefine them. You can use the **-dM (-qshowmacros) -E** compiler options to view the values of the predefined macros.

Table 27. Compiler-related predefined macros

Predefined macro name	Description	Predefined value
__clang__	Indicates that Clang compiler is used.	1
__clang_major__	Indicates the major version number of the Clang compiler.	3
__clang_minor__	Indicates the minor version number of the Clang compiler.	4
__clang_patchlevel__	Indicates the patch level number of the Clang compiler.	0
__clang_version__	Indicates the full version of the Clang compiler.	3.4 (tags/RELEASE_34/final)
__ibmxl__	Indicates the XL C/C++ compiler is being used.	1

__ibmxl_vrm__	Indicates the VRM level of the XL C/C++ compiler using a single integer for sorting purposes.	A hexadecimal integer whose value is as follows:
		`(((__ibmxl_version__) << 24) | \`
		` ((__ibmxl_release__) << 16) | \`
		` ((__ibmxl_modification__) << 8) \`
		`)`

__ibmxl_version__	Indicates the version number of the XL C/C++ compiler.	An integer that represents the version number
__ibmxl_release__	Indicates the release number of the XL C/C++ compiler.	An integer that represents the release number

Table 27. Compiler-related predefined macros (continued)

Predefined macro name	Description	Predefined value
__ibmxl_modification__	Indicates the modification number of the XL C/C++ compiler.	An integer that represents the modification number
__ibmxl_ptf_fix_level__	Indicates the PTF fix level of the XL C/C++ compiler.	An integer that represents the fix number
__llvm__	Indicates that an LLVM backend is used.	1

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 28. Platform-related predefined macros

Predefined macro name	Description	Predefined value	Predefined under the following conditions
_BIG_ENDIAN, __BIG_ENDIAN__	Indicates that the platform is big-endian (that is, the most significant byte is stored at the memory location with the lowest address).	1	Always predefined.
__ELF__	Indicates that the ELF object model is in effect.	1	Always predefined for the Linux platform.
C++ __GXX_WEAK__	Indicates that weak symbols are supported (used for template instantiation by the linker).	1	Always predefined.
__HOS_LINUX__	Indicates that the host operating system is Linux. Protected.	1	Always predefined for all Linux platforms.
__linux, __linux__, linux, __gnu_linux__	Indicates that the platform is Linux.	1	Always predefined for all Linux platforms.
_LP64, __LP64__	Indicates that the target platform uses 64-bit `long int` and pointer types, and a 32-bit `int` type.	1	Predefined when the target platform uses 64-bit `long int` and pointer types, and 32-bit a `int` type.
__THW_BIG_ENDIAN__	Indicates that the target is a big-endian architecture.	1	Always predefined.
__TOS_LINUX__	Indicates that the target operating system is Linux.	1	Predefined when the target OS is a z Systems architecture.
__unix, __unix__, unix	Indicates that the operating system is a variety of UNIX.	1	Always predefined.

Macros related to compiler features

Feature-related macros are predefined according to the setting of specific compiler options or pragmas. Unless noted otherwise, all feature-related macros are protected, 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"
- "Macros related to architecture settings" on page 175
- "Macros related to language levels" on page 176

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 29. General option-related predefined macros

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect
__64BIT__	Indicates that 64-bit compilation mode is in effect.	1	-m64 (-q64)
_CHAR_SIGNED, __CHAR_SIGNED__	Indicates that the default character type is `signed char`.	1	-fsigned-char (-qchars=signed)
_CHAR_UNSIGNED, __CHAR_UNSIGNED__	Indicates that the default character type is `unsigned char`.	1	-funsigned-char (-qchars=unsigned)
C++ __EXCEPTIONS	Indicates that C++ exception handling is enabled.	1	-qeh
__GXX_RTTI	Indicates that runtime type identification (RTTI) information is enabled.	1	-qrtti, -fno-rtti (-qnortti)
C++ __INITAUTO__	Indicates the value to which automatic variables which are not explicitly initialized in the source program are to be initialized.	The two-digit hexadecimal value specified in the **-qinitauto** compiler option.	-qinitauto=*hex value*
C++ __INITAUTO_W__	Indicates the value to which automatic variables which are not explicitly initialized in the source program are to be initialized.	An eight-digit hexadecimal corresponding to the value specified in the **-qinitauto** compiler option repeated 4 times.	-qinitauto=*hex value*

Table 29. General option-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect
C++ __LIBANSI__	Indicates that calls to functions whose names match those in the C Standard Library are in fact the C library functions, enabling certain compiler optimizations.	1	-qlibansi
__LONGDOUBLE128, __LONG_DOUBLE_128__	Indicates that the size of a `long double` type is 128 bits.	1	Always predefined.
__OPTIMIZE__	Indicates the level of optimization in effect.	2	-O | -O2
		3	-O3
__OPTIMIZE_SIZE__	Indicates that optimization for code size is in effect.	1	-O | -O2 | -O3
__RTTI_ALL__	Indicates that runtime type identification (RTTI) information for all operators is enabled.	1	-qrtti
C++ __RTTI_DYNAMIC_CAST__	Indicates that runtime type identification (RTTI) information for the `dynamic_cast` operator is generated.	1	-qrtti
C++ __RTTI_TYPE_INFO__	Indicates that runtime type identification (RTTI) information for the `typeid` operator is generated.	1	-qrtti
C++ __NO_RTTI__	Indicates that runtime type identification (RTTI) information is disabled.	1	-fno-rtti (-qnortti)
__VEC__	Indicates support for vector data types.	10301	-mzvector

Macros related to architecture settings

The following macros can be tested for target architecture settings. All of these macros are predefined to a value of 1 by a **-march** compiler option setting, or any other compiler option that implies that setting. If the **-march** suboption enabling the feature is not in effect, then the macro is undefined.

Table 30. ***-march****-related macros*

Macro name	Description	Predefined by the following -march suboptions
__ARCH_z10	Indicates that the application is targeted to run on IBM System z10 EC and System z10 BC in z/Architecture mode.	z10, arch8
__ARCH_z196	Indicates that the application is targeted to run on IBM zEnterprise 196 and zEnterprise 114 in z/Architecture mode.	z196, arch9
__ARCH_zEC12	Indicates that the application is targeted to run on IBM zEnterprise EC12 and zEnterprise BC12 in z/Architecture mode.	zEC12, arch10
__ARCH_z13	Indicates that the application is targeted to run on IBM z13 in z/Architecture mode.	z13, arch11

Related information

- "-march (-qarch)" on page 92

Macros related to language levels

The following macros except C++ __cplusplus, __STDC__ C++ , and C __STDC_VERSION__ C are predefined to a value of 1 by a specific language level, represented by a suboption of the **-std (-qlanglvl)** compiler option, or any invocation or pragma that implies that suboption. If the suboption enabling the feature is not in effect, then the macro is undefined. For descriptions of the features related to these macros, see the *XL C/C++ Language Reference and the C and C++ language standards*.

Table 31. Predefined macros for language features

Predefined macro name	Description	Predefined when the following language level is in effect
C++ __BOOL__	Indicates that the `bool` keyword is accepted.	Always defined.
C++ __cplusplus	The numeric value that indicates the supported language standard as defined by that specific standard.	The format is *yyyymm*L. (For example, the format is 199901L for C99.)
C++ __IBMCPP_COMPLEX_INIT	Indicates support for the initialization of complex types: float _Complex, double _Complex, and long double _Complex.	extended
__STDC__	Indicates that the compiler conforms to the ANSI/ISO C standard.	C Predefined to 1 if ANSI/ISO C standard conformance is in effect. C++ Explicitly defined to 0.

Table 31. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
__STDC_HOSTED__	Indicates that the implementation is a hosted implementation of the ANSI/ISO C standard. (That is, the hosted environment has all the facilities of the standard C available).	C stdc11 | extc1x | stdc99 | extc99 C++ extended0x
C11 __STDC_NO_ATOMICS__	Indicates that the implementation does not have the full support of the atomics feature.	stdc11 | extc1x
C11 __STDC_NO_THREADS__	Indicates that the implementation does not have the full support of the threads feature.	stdc11 | extc1x
C __STDC_VERSION__	Indicates the version of ANSI/ISO C standard which the compiler conforms to.	The format is *yyyymm*L. (For example, the format is 199901L for C99.)

Unsupported macros from other XL compilers

The following macros, which might be supported by other XL compilers, are unsupported in IBM XL C/C++ for Linux on z Systems, V1.2. You can specify the **-Wunsupported-xl-macro** option to check whether any unsupported macro is used; if an unsupported macro is used, the compiler issues a warning message.

You might want to edit your source code to remove references of the unsupported macros during compiler migration.

Table 32. Unsupported macros indicating the XL C/C++ compiler product

__IBMC__	__xlc__
__IBMCPP__	__xlC__
	__xlC_ver__

Table 33. Unsupported macros that are related to the platform

_LITTLE_ENDIAN, __LITTLE_ENDIAN__
_ILP32, __ILP32__
__POWERPC__
__PPC__
__PPC64__
__THW_PPC__

Table 34. Unsupported macros related to compiler option settings

__ALTIVEC__
__LONGDOUBLE64
__IBM_GCC_ASM
__IBM_STDCPP_ASM
__IGNERRNO__
__TEMPINC__

Table 35. Unsupported macros related to architecture settings

_ARCH_PPC
_ARCH_PPC64
_ARCH_PPCGR
_ARCH_PWR4
_ARCH_PWR5
_ARCH_PWR5X
_ARCH_PWR6
_ARCH_PWR6E
_ARCH_PWR7
_ARCH_PWR8
__ARCH_z10
__ARCH_z196
__ARCH_zEC12

Table 36. Unsupported macros related to language levels

__C99_BOOL	__IBM_DOLLAR_IN_ID
__C99_COMPLEX	__IBM_EXTENSION_KEYWORD
__C99_COMPOUND_LITERAL	__IBM_GCC__INLINE__
__C99_CPLUSCMT	__IBM_GENERALIZED_LVALUE
__C99_DESIGNATED_INITIALIZER	__IBM_INCLUDE_NEXT
__C99_DUP_TYPE_QUALIFIER	__IBM_LABEL_VALUE
__C99_EMPTY_MACRO_ARGUMENTS	__IBM_LOCAL_LABEL
__C99_FLEXIBLE_ARRAY_MEMBER	__IBM_MACRO_WITH_VA_ARGS
__C99_FUNC__	__IBM_NESTED_FUNCTION
__C99_HEX_FLOAT_CONST	__IBM_PP_PREDICATE
__C99_INLINE	__IBM_PP_WARNING
__C99_LLONG	__IBM_REGISTER_VARS
__C99_MACRO_WITH_VA_ARGS	__IBM__TYPEOF__
__C99_MAX_LINE_NUMBER	__IBMC_COMPLEX_INIT
__C99_MIXED_DECL_AND_CODE	__IBMC_GENERIC
__C99_MIXED_STRING_CONCAT	__IBMC_NORETURN
__C99_NON_LVALUE_ARRAY_SUB	__IBMC_STATIC_ASSERT
__C99_NON_CONST_AGGR_INITIALIZER	__IBMCPP_AUTO_TYPEDEDUCTION
__C99_PRAGMA_OPERATOR	__IBMCPP_C99_LONG_LONG
__C99_REQUIRE_FUNC_DECL	__IBMCPP_C99_PREPROCESSOR
__C99_RESTRICT	__IBMCPP_CONSTEXPR
__C99_STATIC_ARRAY_SIZE	__IBMCPP_DECLTYPE
__C99_STD_PRAGMAS	__IBMCPP_DELEGATING_CTORS
__C99_TGMATH	__IBMCPP_EXPLICIT_CONVERSION_OPERATORS
__C99_UCN	__IBMCPP_EXTENDED_FRIEND
__C99_VAR_LEN_ARRAY	__IBMCPP_EXTERN_TEMPLATE
__C99_VARIABLE_LENGTH_ARRAY	__IBMCPP_INLINE_NAMESPACE
__DIGRAPHS__	__IBMCPP_REFERENCE_COLLAPSING
__EXTENDED__	__IBMCPP_RIGHT_ANGLE_BRACKET
__IBM__ALIGN	__IBMCPP_RVALUE_REFERENCES
__IBM__ALIGNOF__	__IBMCPP_SCOPED_ENUM
__IBM_ALIGNOF__	__IBMCPP_STATIC_ASSERT
__IBM_ATTRIBUTES	__IBMCPP_UNIFORM_INIT
__IBM_COMPUTED_GOTO	__IBMCPP_VARIADIC_TEMPLATES
	_LONG_LONG

Chapter 6. Compiler built-in functions

A built-in function is a coding extension to C and C++ that allows a programmer to use the syntax of C function calls and C variables to access the instruction set of the processor of the compiling machine. IBM z Systems architectures have special instructions that enable the development of highly optimized applications. Access to some z Systems instructions cannot be generated using the standard constructs of the C and C++ languages. Other instructions can be generated through standard constructs, but using built-in functions allows exact control of the generated code. Inline assembly language programming, which uses these instructions directly, is fully supported. Furthermore, the technique can be time-consuming to implement.

As an alternative to managing hardware registers through assembly language, XL C/C++ built-in functions provide access to the optimized z Systems instruction set and allow the compiler to optimize the instruction scheduling.

C++ To call any of the XL C/C++ built-in functions in C++, you must include the header file `builtins.h` in your source code. C++

The following sections describe the available built-in functions for the z Systems platform.

Fixed-point built-in functions

Fixed-point built-in functions are grouped into the following categories:
- "Absolute value functions"
- "Population count functions"

Absolute value functions

__labs, __llabs

Purpose

Absolute Value Long, Absolute Value Long Long

Returns the absolute value of the argument.

Prototype

signed long __labs (signed long);

signed long long __llabs (signed long long);

Population count functions

__popcnt4, __popcnt8

Purpose

Population Count, 4-byte or 8-byte integer

Returns the number of bits set for a 32-bit or 64-bit integer.

© Copyright IBM Corp. 2015

Prototype

int __popcnt4 (unsigned int);

int __popcnt8 (unsigned long long);

Cache-related built-in functions

Cache-related built-in functions fall into the "Data cache functions" category.

Data cache functions

__dcbf

Purpose

Releases the cache line that contains the specified address *Op2* from all accesses.

Prototype

void __dcbf(const void* *Op2*)

__dcbst

Purpose

Releases the cache line that contains the specified address *Op2* from store access and retains the data in the cache line for fetch access.

Prototype

void __dcbst(const void* *Op2*)

__dcbt

Purpose

Prefetches the cache line that contains the specified address *Op2* into the cache for fetch access.

Prototype

void __dcbt(const void* *Op2*)

__dcbtst

Purpose

Prefetches the cache line that contains the specified address *Op2* into the cache for store access.

Prototype

void __dcbtst (const void* *Op2*)

Block-related built-in functions

bzero

Purpose

Sets the first *n* bytes of the byte area starting at *s* to zero.

Prototype

void bzero(void* *s*, size_t *n*);

Parameters

n The size of the data.

s The starting address in the byte area.

Vector built-in functions

You can use vector built-in functions to access and manipulate individual elements of vectors when the following conditions are met:

- The compiler runs on a Linux distribution that has vector support.
- Both the **-mzvector** option and the **-march=z13** option, or its equivalent, are in effect.
- The `builtins.h` or `vecintrinc.h` header file is included.

For detailed information about individual vector built-in functions, see Vector built-in functions in the *XL C/C++ Optimization and Programming Guide*.

GCC atomic memory access built-in functions (IBM extension)

This section provides reference information for atomic memory access built-in functions whose behavior corresponds to that provided by GNU Compiler Collection (GCC). In a program with multiple threads, you can use these functions to atomically and safely modify data in one thread without interference from other threads.

These built-in functions manipulate data atomically, regardless of how many processors are installed in the host machine.

In the prototype of each function, the parameter types *T*, *U*, and *V* can be of pointer or integral type. *U* and *V* can also be of real floating-point type, but only when *T* is of integral type. The following tables list the integral and floating-point types that are supported by these built-in functions.

Table 37. Supported integral data types

`signed char`	`unsigned char`
`short int`	`unsigned short int`
`int`	`unsigned int`
`long int`	`unsigned long int`
`long long int` 1	`unsigned long long int` 1
C++ `bool`	C `_Bool`

Table 37. Supported integral data types (continued)

1 **Restriction:** This type is supported only on 64-bit platforms.

Table 38. Supported floating-point data types

float	double
long double	

In the prototype of each function, the ellipsis (...) represents an optional list of parameters. XL C/C++ ignores these optional parameters and protects all globally accessible variables.

The GCC atomic memory access built-in functions are grouped into the following categories.

Atomic lock, release, and synchronize functions

__sync_lock_test_and_set

Purpose

This function atomically assigns the value of *__v* to the variable that *__p* points to.

An acquire memory barrier is created when this function is invoked.

Prototype

T __sync_lock_test_and_set (*T** *__p*, *U* *__v*, ...);

Parameters

__p
The pointer of the variable that is to be set.

__v
The value to set to the variable that *__p* points to.

Return value

The function returns the initial value of the variable that *__p* points to.

__sync_lock_release

Purpose

This function releases the lock acquired by the `__sync_lock_test_and_set` function, and assigns the value of zero to the variable that *__p* points to.

A release memory barrier is created when this function is invoked.

Prototype

void __sync_lock_release (*T** *__p*, ...);

Parameters

__p
The pointer of the variable that is to be set.

__sync_synchronize

Purpose

This function synchronizes data in all threads.

A full memory barrier is created when this function is invoked.

Prototype

void __sync_synchronize ();

Atomic fetch and operation functions

__sync_fetch_and_add

Purpose

This function atomically adds the value of *__v* to the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_fetch_and_add (*T** *__p*, *U* *__v*, ...);

Parameters

__p
: The pointer of a variable to which *__v* is to be added. The value of this variable is to be changed to the result of the add operation.

__v
: The variable whose value is to be added to the variable that *__p* points to.

Return value

The function returns the initial value of the variable that *__p* points to.

__sync_fetch_and_and

Purpose

This function performs an atomic bitwise AND operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_fetch_and_and (*T** *__p*, *U* *__v*, ...);

Parameters

__p
: The pointer of a variable on which the bitwise AND operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
: The variable with which the bitwise AND operation is to be performed.

Return value

The function returns the initial value of the variable that *__p* points to.

__sync_fetch_and_nand

Purpose

This function performs an atomic bitwise NAND operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_fetch_and_nand (*T** *__p*, *U* *__v*, ...);

Parameters

__p
: The pointer of a variable on which the bitwise NAND operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
: The variable with which the bitwise NAND operation is to be performed.

Return value

The function returns the initial value of the variable that *__p* points to.

__sync_fetch_and_or

Purpose

This function performs an atomic bitwise inclusive OR operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_fetch_and_or (*T** *__p*, *U* *__v*, ...);

Parameters

__p
: The pointer of a variable on which the bitwise inclusive OR operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
: The variable with which the bitwise inclusive OR operation is to be performed.

Return value

The function returns the initial value of the variable that *__p* points to.

__sync_fetch_and_sub

Purpose

This function atomically subtracts the value of *__v* from the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_fetch_and_sub (*T** *__p*, *U* *__v*, ...);

Parameters

__p
: The pointer of a variable from which *__v* is to be subtracted. The value of this variable is to be changed to the result of the sub operation.

__v
: The variable whose value is to be subtracted from the variable that *__p* points to.

Return value

The function returns the initial value of the variable that *__p* points to.

__sync_fetch_and_xor

Purpose

This function performs an atomic bitwise exclusive OR operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_fetch_and_xor (*T** *__p*, *U* *__v*, ...);

Parameters

__p
: The pointer of a variable on which the bitwise exclusive OR operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
: The variable with which the bitwise exclusive OR operation is to be performed.

Return value

The function returns the initial value of the variable that *__p* points to.

Atomic operation and fetch functions

__sync_add_and_fetch

Purpose

This function atomically adds the value of *__v* to the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_add_and_fetch (*T** *__p*, *U* *__v*, ...);

Parameters

__p
: The pointer of a variable to which *__v* is to be added. The value of this variable is to be changed to the result of the add operation.

__v
: The variable whose value is to be added to the variable that *__p* points to.

Return value

The function returns the new value of the variable that *__p* points to.

__sync_and_and_fetch

Purpose

This function performs an atomic bitwise AND operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_and_and_fetch (*T** *__p*, *U* *__v*, ...);

Parameters

__p
: The pointer of a variable on which the bitwise AND operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
: The variable with which the bitwise AND operation is to be performed.

Return value

The function returns the new value of the variable that *__p* points to.

__sync_nand_and_fetch

Purpose

This function performs an atomic bitwise NAND operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_nand_and_fetch (*T** *__p*, *U* *__v*, ...);

Parameters

__p
: The pointer of a variable on which the bitwise NAND operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
: The variable with which the bitwise NAND operation is to be performed.

Return value

The function returns the new value of the variable that *__p* points to.

__sync_or_and_fetch

Purpose

This function performs an atomic bitwise inclusive OR operation on the variable *__v* with variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_or_and_fetch (*T** *__p*, *U* *__v*, ...);

Parameters

__p
: The pointer of a variable on which the bitwise inclusive OR operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
: The variable with which the bitwise inclusive OR operation is to be performed.

Return value

The function returns the new value of the variable that *__p* points to.

__sync_sub_and_fetch

Purpose

This function atomically subtracts the value of *__v* from the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_sub_and_fetch (*T** *__p*, *U* *__v*, ...);

Parameters

__p
: The pointer of a variable from which *__v* is to be subtracted. The value of this variable is to be changed to the result of the sub operation.

__v
: The variable whose value is to be subtracted from the variable that *__p* points to.

Return value

The function returns the new value of the variable that *__p* points to.

__sync_xor_and_fetch

Purpose

This function performs an atomic bitwise exclusive OR operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_xor_and_fetch (*T** *__p*, *U* *__v*, ...);

Parameters

__p
: The pointer of the variable on which the bitwise exclusive OR operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
: The variable with which the bitwise exclusive OR operation is to be performed.

Return value

The function returns the new value of the variable that *__p* points to.

Atomic compare and swap functions

__sync_bool_compare_and_swap

Purpose

This function compares the value of *__compVal* with the value of the variable that *__p* points to. If they are equal, the value of *__exchVal* is stored in the address that is specified by *__p*; otherwise, no operation is performed.

A full memory barrier is created when this function is invoked.

Prototype

bool __sync_bool_compare_and_swap (*T* __p*, *U __compVal*, *V __exchVal*, ...);

Parameters

__p
The pointer to a variable whose value is to be compared with.

__compVal
The value to be compared with the value of the variable that *__p* points to.

__exchVal
The value to be stored in the address that *__p* points to.

Return value

If the value of *__compVal* and the value of the variable that *__p* points to are equal, the function returns `true`; otherwise, it returns `false`.

__sync_val_compare_and_swap

Purpose

This function compares the value of *__compVal* to the value of the variable that *__p* points to. If they are equal, the value of *__exchVal* is stored in the address that is specified by *__p*; otherwise, no operation is performed.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_val_compare_and_swap (*T* __p*, *U __compVal*, *V __exchVal*, ...);

Parameters

__p
The pointer to a variable whose value is to be compared with.

__compVal
The value to be compared with the value of the variable that *__p* points to.

__exchVal
The value to be stored in the address that *__p* points to.

Return value

The function returns the initial value of the variable that *__p* points to.

Miscellaneous built-in functions

Miscellaneous functions are grouped into the following categories:

- "Optimization-related functions"
- "Memory-related functions"

Optimization-related functions

__builtin_expect

Purpose

Indicates that an expression is likely to evaluate to a specified value. The compiler may use this knowledge to direct optimizations.

Prototype

long __builtin_expect (long *expression*, long *value*);

Parameters

expression
: Should be an integral-type expression.

value
: Must be a constant literal.

Usage

If the *expression* does not actually evaluate at run time to the predicted value, performance may suffer. Therefore, this built-in function should be used with caution.

Memory-related functions

__alloca

Purpose

Allocates space for an object. The allocated space is put on the stack and freed when the calling function returns.

Prototype

void* __alloca (size_t *size*)

Parameters

size
: An integer representing the amount of space to be allocated, measured in bytes.

__builtin_frame_address, __builtin_return_address

Purpose

Returns the address of the stack frame, or return address, of the current function, or of one of its callers.

Prototype

void* __builtin_frame_address (unsigned int *level*);

void* __builtin_return_address (unsigned int *level*);

Parameters

level
: A constant literal indicating the number of frames to scan up the call stack. The *level* must range from 0 to 63. A value of 0 returns the frame or return address of the current function, a value of 1 returns the frame or return address of the caller of the current function and so on.

Return value

Returns 0 when the top of the stack is reached. Optimizations such as inlining may affect the expected return value by introducing extra stack frames or fewer stack frames than expected. If a function is inlined, the frame or return address corresponds to that of the function that is returned to.

Hardware built-in functions

General hardware built-in functions provide access to general purpose instructions that are not normally generated by the compiler. For more information on these instructions, see z/OS Architecture Principle of Operations.

You must include the `builtins.h` header file before using any of these hardware built-in functions.

Purpose

Compares two signed packed decimal integers.

Prototypes

int __cp (unsigned char **Op1*, unsigned char *Len1*, unsigned char **Op2*, unsigned char *Len2*);

Parameter

Op1
: The address of the first operand.

Len1
: The number of additional bytes to the left of the first operand.

: The value must be in the range 0 - 15 inclusive.

Op2
: The address of the second operand.

Len2
: The number of additional bytes to the left of the second operand.

: The value must be in the range 0 - 15 inclusive.

Usage

The function compares the first operand *Op1* with the second operand *Op2*, and the result is indicated in the returned condition code.

Note: When either *Len1* or *Len2* is not specified as a literal, an EX instruction is generated to execute a target CP instruction with the length encoded in the register that is used by the EX instruction.

The preferred and alternate sign codes for a particular sign are treated as equivalent for comparison purposes.

A negative zero and a positive zero are considered equal.

Return value

The function returns the condition code set by the CP instruction.

Table 39. Resulting condition code

Code	Description
0	The first operand is equal to the second operand.
1	The first operand is lower than the second operand.
2	The first operand is higher than the second operand.

Related information:

z/OS Architecture Principle of Operations

__cvb

Purpose

Converts an 8-byte, signed packed decimal integer to a signed binary integer and loads the result into a general register.

After the conversion operation is completed, the number is in the proper form for use as an operand in signed binary arithmetic.

Prototypes

int __cvb (char **Op2*);

Parameter

Op2
A pointer to an 8-byte storage area that contains a valid signed packed decimal integer to be converted to a signed binary integer.

Return value

The function returns the converted 32-bit signed binary integer.

Related information:

z/OS Architecture Principle of Operations

__cvbg

Purpose

Converts a packed decimal integer at a given storage area to a 64-bit signed binary integer.

Prototypes

long long __cvbg (char **Op2*);

Parameter

Op2
: A pointer to a 16-byte storage area that contains a packed decimal integer

Return value

The function returns the converted 64-bit signed binary integer.

Related information:

z/OS Architecture Principle of Operations

__cvd

Purpose

Converts a signed binary integer to a packed decimal integer and places the result at a given address.

Prototypes

void __cvd (int *Op1*, char **Op2*);

Parameter

Op1
: A 32-bit signed binary integer to be converted to a packed decimal integer

Op2
: A pointer to an 8-byte storage area that receives the converted packed decimal integer

Return value

The function does not return any value.

Related information:

z/OS Architecture Principle of Operations

__cvdg

Purpose

Converts a 64-bit signed binary integer to a packed decimal integer and saves the result to a given storage area.

Prototypes

void __cvdg (long long *Op1*, char **Op2*);

Parameter

Op1
: A 64-bit signed binary integer to be converted into a packed decimal integer

Op2
: A pointer to a 16-byte storage area that receives the converted packed decimal integer

Return value

The function does not return any value.

Related information:

z/OS Architecture Principle of Operations

__dp

Purpose

Divides a signed packed decimal integer (the dividend) by another signed packed decimal integer (the divisor).

Prototypes

void __dp (unsigned char **Op1*, unsigned char *Len1*, unsigned char **Op2*, unsigned char *Len2*);

Parameter

Op1
: The address of the first operand, the dividend.

  The dividend cannot exceed 31 digits and sign.

Len1
: The number of additional bytes to the left of the dividend.

  The value must be greater than *Len2*.

Op2
: The address of the second operand, the divisor.

  The divisor cannot exceed 15 digits and sign.

Len2
: The number of additional bytes to the left of the divisor.

  The value must be in the range 0 - 7 inclusive and less than *Len1*.

Usage

The first operand *Op1* is divided by the second operand *Op2*. The resulting quotient and remainder are placed at the location of the dividend. The quotient is placed leftmost at the dividend location. The length of the quotient field is equal to the difference between the dividend length and the divisor length (*Len2* – *Len1*). The remainder is placed rightmost at the dividend location and has a length equal to the divisor length.

Note: When either *Len1* or *Len2* is not specified as a literal, an EX instruction is generated to execute a target DP instruction with the length encoded in the register that is used by the EX instruction.

Return value

The function does not return any value.

Related information:

z/OS Architecture Principle of Operations

__fidbr

Purpose

Rounds a long binary-floating-point number to an integer value in the same floating-point format using the specified rounding mode.

Prototypes

double __fidbr (int *M3*, double *Op2*)

Note: *M3* must be a literal.

Parameter

M3 The 4-bit mask to select the rounding mode

Op2
A long binary-floating-point number

Usage

This function takes effect only when the **-qfloat** option is enabled.

The following table describes the rounding methods that are represented by each *M3* modifier.

M3	**Rounding method**
0	According to the current binary-floating-point rounding mode
1	Rounds to nearest with ties away from zero
3	Rounds to prepare for shorter precision
4	Rounds to nearest with ties to even
5	Rounds toward zero
6	Rounds toward positive infinity
7	Rounds toward negative infinity
Note: *M3* is valid only when the modifier value is 0, 1, or 3 - 7. *M3* is also invalid when the floating-point extension facility is not installed and the modifier value is 3.	

Return value

The function returns an integer value in the floating-point format.

Related information

- "__fiebr"
- "__fixbr" on page 197
- -qfloat
- z/Architecture Principles of Operation
- z/OS XL C/C++ Programming Guide

__fiebr

Purpose

Rounds a short binary-floating-point number to an integer value in the same floating-point format using the specified rounding mode.

Prototypes

float __fiebr (int *M3*, float *Op2*)

Note: *M3* must be a literal.

Parameter

M3 The 4-bit mask to select the rounding mode

Op2
A short binary-floating-point number

Usage

This function takes effect only when the **-qfloat** option is in effect.

The following table describes the rounding methods that are represented by each *M3* modifier.

M3	**Rounding method**
0	According to the current binary-floating-point rounding mode
1	Rounds to nearest with ties away from zero
3	Rounds to prepare for shorter precision
4	Rounds to nearest with ties to even
5	Rounds toward zero
6	Rounds toward positive infinity
7	Rounds toward negative infinity
Note: *M3* is valid only when the modifier value is 0, 1, or 3 - 7. *M3* is also invalid when the floating-point extension facility is not installed and the modifier value is 3.	

Return value

The function returns an integer value in the floating-point format.

Related information

- "__fidbr" on page 195
- "__fixbr" on page 197

- -qfloat
- z/OS XL C/C++ Programming Guide

__fixbr

Purpose

Rounds an extended binary-floating-point number to an integer value in the same floating-point format using the specified rounding mode.

Prototypes

long double __fixbr (int *M3*, long double *Op2*)

Note: *M3* must be a literal.

Parameter

M3 The 4-bit mask to select the rounding mode

Op2
An extended binary-floating-point number

Usage

This function takes effect only when the **-qfloat** option is enabled.

The following table describes the rounding methods that are represented by each *M3* modifier.

M3	**Rounding method**
0	According to the current binary-floating-point rounding mode
1	Rounds to nearest with ties away from zero
3	Rounds to prepare for shorter precision
4	Rounds to nearest with ties to even
5	Rounds toward zero
6	Rounds toward positive infinity
7	Rounds toward negative infinity
Note: *M3* is valid only when the modifier value is 0, 1, or 3 - 7. *M3* is also invalid when the floating-point extension facility is not installed and the modifier value is 3.	

Return value

The function returns an integer value in the floating-point format.

Related information

- "__fidbr" on page 195
- "__fiebr" on page 196
- -qfloat
- z/Architecture Principles of Operation
- z/OS XL C/C++ Programming Guide

__lcbb

Purpose

Counts the number of bytes from a given address without crossing the specified block boundary.

Prototypes

unsigned int __lcbb (const void **Op2*, unsigned short *boundary*)

Parameter

Op2
: The address of the second operand in the hardware instruction.

boundary
: The boundary to encode in the hardware instruction. It must be a literal whose value is 64, 128, 256, 512, 1024, 2048, or 4096.

Usage

This instruction can be used only on IBM z13 models.

If the return value is 16, the LCBB instruction sets the condition code to 0. If the return value is less than 16, the LCBB instruction sets the condition code to 3.

Return value

The function returns the first operand that is set by the LCBB instruction. The first operand contains the number of bytes to load from the second operand location without crossing the specified block boundary. If the number of bytes is greater than 16, the first operand is set to 16.

Related information

- z/OS Architecture Principle of Operations

__mp

Purpose

Multiplies a signed packed decimal integer (the multiplicand) by another signed packed decimal integer (the multiplier).

Prototypes

void __mp (unsigned char **Op1*, unsigned char *Len1*, unsigned char **Op2*, unsigned char *Len2*);

Parameter

Op1
: The address of the multiplicand.

Len1
: The number of additional bytes to the left of the multiplicand.

  The value must be greater than *Len2*.

Op2
: The address of the multiplier.

Len2
: The number of additional bytes to the left of the multiplier.

  The value must be in the range 0 - 7 inclusive and less than *Len1*.

Usage

The multiplicand must have at least as many bytes of leftmost zeros as the number of bytes in the multiplier.

The multiplier length cannot exceed 15 digits and sign, and it must be less than the multiplicand length.

Note: When either *Len1* or *Len2* is not specified as a literal, an EX instruction is generated to execute a target MP instruction with the length encoded in the register that is used by the EX instruction.

The product is placed at the location of the multiplicand.

The product cannot exceed 31 digits and sign. The leftmost digit of the product is always zero.

Return value

The function does not return any value.

Related information:

z/OS Architecture Principle of Operations

__srp

Purpose

Shifts a signed packed decimal integer with or without rounding.

Prototypes

int __srp (unsigned char **Op1*, unsigned char *Len1*, signed char *Op2*, unsigned char *Op3*);

Parameter

Op1
: The address of the first operand, which points to a signed packed decimal integer to be shifted with or without rounding.

Len1
: The number of additional bytes to the left of the first operand.

  The value must be in the range 0 - 15 inclusive.

Op2
: The address of the second operand, which points to a 6-bit signed binary integer.

  Op2 indicates the direction and the number of decimal digit positions to be shifted.

The value must be in the range -32 to 31 inclusive. Positive values specify shifting to the left. Negative values specify shifting to the right.

Op3
The address of the third operand, which points to a decimal rounding digit.

Usage

The function can be used for shifting up to 31 digit positions left and up to 32 digit positions right. Only the digit portion of the first operand *Op1* is shifted; the sign position does not participate in the shifting.

Bits 58-63 of *Op2* are the shift value, and the leftmost bits are ignored.

For a right shift, that is, when *Op2* is negative, the absolute value of the first operand *Op1* is rounded to the rounding digit *Op3*. The rounding digit 5 provides conventional rounding of the result. The rounding digit 0 specifies truncation without rounding.

The result is placed at the location of the first operand. Zeros fill the vacated digits.

Note: When either *Len1* or *Op3* is not specified as a literal, an EX instruction is generated to execute a target SRP instruction with the *Len1* or *Op3* parameter encoded in the register that is used by the EX instruction.

Return value

The function returns the condition code set by the SRP instruction.

Table 40. Resulting condition code

Code	Description
0	The result is zero. No overflow occurs.
1	The result is less than zero. No overflow occurs.
2	The result is greater than zero. No overflow occurs.
3	Overflow occurs.

Related information:

z/OS Architecture Principle of Operations

__stck

Purpose

Stores the internal 8-byte time-of-day (TOD) clock to the specified memory location.

Prototypes

int __stck (unsigned long long **Op1*);

Parameter

Op1
The address of the memory location to store the TOD clock

Usage

A serialization function is performed before the value of the TOD clock is fetched and stored.

Return value

The function returns the condition code.

Related information

- "__stckf"
- z/OS Architecture Principle of Operations

__stckf

Purpose

Stores the internal 8-byte time-of-day (TOD) clock to the specified memory location.

Prototypes

int __stckf (unsigned long long **Op1*);

Parameter

Op1
: The address of the memory location to store the TOD clock

Usage

The `__stckf` built-in function runs faster than the `__stck` built-in function because other processors are not serialized before the value of the TOD clock is fetched and stored.

Return value

The function returns the condition code.

Related information

- "__stck" on page 200
- z/OS Architecture Principle of Operations

__zap

Purpose

Places a signed packed decimal integer at a given address. The operation is equivalent to an addition to zero.

Prototypes

int __zap (unsigned char **Op1*, unsigned char *Len1*, unsigned char **Op2*, unsigned char *Len2*);

Parameter

Op1
The address of the first operand, which is to receive the result.

Len1
The number of additional bytes to the left of the sign byte of the first operand.

The value must be in the range 0 - 15 inclusive.

Op2
The address of the second operand, which points to a signed packed decimal integer to be placed at the location specified by *Op1*.

Len2
The number of additional bytes to the left of the sign byte of the second operand.

The value must be in the range 0 - 15 inclusive.

Usage

This function places the second operand *Op2* at the location specified by the first operand *Op1*.

If the first operand is too short to contain all leftmost nonzero digits of the second operand, decimal overflow occurs and the operation is completed. The result is obtained by ignoring the overflow digits. If the decimal-overflow mask is one, a program interruption for decimal overflow occurs.

If overflow does not occur, the sign of a zero result is made positive. If overflow occurs, a zero result is given the sign of the second operand but with the preferred sign code.

If the two operands overlap and the rightmost byte of the first operand is exactly or to the right of the rightmost byte of the second operand, the result is obtained as if the operands were processed right to left.

If the two operands overlap and the rightmost byte of the first operand is to the left of the rightmost byte of the second operand, either of the following consequences might occur depending on the model:
- A data exception is recognized.
- The result is obtained as if the entire second operand were fetched before any byte of the result is stored.

Note: When either *Len1* or *Len2* is not specified as a literal, an EX instruction is generated to execute a target ZAP instruction with the length encoded in the register that is used by the EX instruction.

Return value

The function returns the condition code set by the ZAP instruction.

Table 41. Resulting condition code

Code	Description
0	The result is zero. No overflow occurs.
1	The result is less than zero. No overflow occurs.

Table 41. Resulting condition code (continued)

Code	Description
2	The result is greater than zero. No overflow occurs.
3	Overflow occurs.

Related information:

z/OS Architecture Principle of Operations

Transactional memory built-in functions

Transactional memory is a model for parallel programming. This module provides functions that allow you to designate a block of instructions or statements to be treated atomically. Such an atomic block is called a transaction. When a thread executes a transaction, all of the memory operations within the transaction occur simultaneously from the perspective of other threads.

For some kinds of parallel programs, a transaction implementation can be more efficient than other implementation methods, such as locks. You can use these built-in functions to mark the beginning and end of transactions, and to diagnose the reasons for failure.

In the transactional memory built-in functions, the *TM_buff* parameter allows for a user-provided memory location to be used to store the transaction state and debugging information.

The transactional state is entered following a successful call to `__TM_begin` or `__TM_simple_begin`, and ended by `__TM_end`, `__TM_abort`, `__TM_named_abort`, or by transaction failure.

Transaction failure occurs when any of the following conditions is met:

- Memory that is accessed in the transactional state is accessed by another thread or by the same thread running in the suspended state before the transaction completes.
- The architecture-defined footprint for memory accesses within a transaction is exceeded.
- The architecture-defined nesting limit for nested transactions is exceeded.

Transactions can be nested. You can use `__TM_begin` or `__TM_simple_begin` in the transactional state. Within an outermost transaction initiated with `__TM_begin`, nested transactions must be initiated with `__TM_simple_begin`, or by `__TM_begin` using the same buffer of the outermost containing transaction.

A nested transaction is subsumed into the containing transaction. Therefore, a failure of the nested transaction is treated as a failure of all containing transactions, and the nested transaction completes only when all contained transactions complete.

Note: You must include the `htmxlintrin.h` file in the source code if you use any of the transactional memory built-in functions.

Transaction begin and end functions

__TM_begin

Purpose

Marks the beginning of an expensive transaction, which provides full debugging capability.

Prototype

long __TM_begin (void* const *TM_buff*);

Parameter

TM_buff
: The address of a 256-byte transaction diagnostic block (TDB) that contains diagnostic information.

Usage

Upon a transaction failure (including a user abort), execution resumes from the point immediately following the `__TM_begin` that initiated the failed transaction as if the `__TM_begin` were unsuccessful.

You can use the transaction inquiry functions to query the transaction status.

If the transaction fails, the TDB is populated with lots of debug information.

Note: Using this built-in function causes the compiler to save all variables that are kept in GPRs. However, variables that are kept in FPRs or ARs are not saved. Saving variables of the `float` type is not supported, and the facility of filtering some interrupts is not provided.

Return value

This function returns _HTM_TBEGIN_STARTED if successful; otherwise, it returns a different value.

Related information

- "__TM_simple_begin" on page 205
- "Transaction inquiry functions" on page 206

- z/OS Architecture Principle of Operations

__TM_end

Purpose

Marks the end of a transaction.

Prototype

long __TM_end ();

Return value

The return value is _HTM_TBEGIN_STARTED if the thread is in the transactional state before the instruction starts; otherwise, it returns a different value.

Related information

- z/OS Architecture Principle of Operations

__TM_simple_begin

Purpose

Marks the beginning of a cheap transaction. Compared with the `__TM_begin` built-in function, `__TM_simple_begin` has better performance but provides limited debugging capability.

Prototype

long __TM_simple_begin ();

Usage

Upon a transaction failure (including a user abort), execution resumes from the point immediately following the `__TM_simple_begin` function that initiated the failed transaction as if the `__TM_simple_begin` were unsuccessful.

The transaction status of transactions started using `__TM_simple_begin` cannot be queried by using the transaction inquiry functions.

A cheap transaction has better performance than an expensive transaction. However, if a cheap transaction fails, the only failure information available is the 2-bit *cc* returned by the `tbegin` hardware instruction.

Note: Using this built-in function causes the compiler to save all variables that are kept in GPRs. However, variables that are kept in FPRs or ARs are not saved. Saving variables of the `float` type is not supported, and the facility of filtering some interrupts is not provided.

Return value

This function returns _HTM_TBEGIN_STARTED if successful; otherwise, it returns a different value.

Related information

- "__TM_begin" on page 204
- "Transaction inquiry functions" on page 206
- z/OS Architecture Principle of Operations

Transaction abort functions

__TM_abort

Purpose

Aborts a transaction with failure code 0.

Prototype

```
void __TM_abort ();
```

Related information

- "__TM_named_abort"
- z/OS Architecture Principle of Operations

__TM_abort_assist

Purpose

Requests the processor to increase the likelihood of successful completion of the transaction in a subsequent execution. The processor takes escalating actions based on the number of times that the transaction has been aborted.

Prototypes

void __TM_abort_assist (unsigned int *op1*);

Parameter

op1
: The number of times that the transaction has been aborted.

Related information

- z/OS Architecture Principle of Operations

__TM_named_abort

Purpose

Aborts a transaction with the specified failure code.

Prototype

void __TM_named_abort (unsigned char const *code*);

Parameter

code
: The specified failure code. It is a literal that is in the range of 0 - 255.

Related information

- "__TM_abort" on page 205
- z/OS Architecture Principle of Operations

Transaction inquiry functions

__TM_failure_address

Purpose

Gets the code address at which the most recent transaction was aborted.

Prototypes

long __TM_failure_address (void* const *TM_buff*);

Parameter

TM_buff
: The address of a 256-byte transaction diagnostic block (TDB) that contains diagnostic information.

Return value

This function returns the address at which the most recent transaction was aborted. In 64-bit mode, the address is obtained from the fourth doubleword of the TDB. In 31-bit mode, the address is obtained from the righmost word of the TDB.

Related information

- z/OS Architecture Principle of Operations

__TM_failure_code

Purpose

Provides the raw failure code for the transaction.

Prototypes

long long __TM_failure_code (void* const *TM_buff*);

Parameter

TM_buff
: The address of a 256-byte transaction diagnostic block (TDB) that contains diagnostic information.

Return value

The function returns the raw failure code for the transaction. The raw failure code is obtained from byte 8 - 15 of the TDB.

Related information

- z/OS Architecture Principle of Operations

__TM_is_conflict

Purpose

Queries whether the transaction was aborted because of a conflict.

Prototypes

long __TM_is_conflict (void* const *TM_buff*);

Parameter

TM_buff
: The address of a 256-byte transaction diagnostic block (TDB) that contains diagnostic information.

Return value

This function returns 1 if both of the following qualifications are met; otherwise, it returns 0:

- The TDB is valid.
- The transaction was aborted because of a conflict. The transaction abort code, which is the second word of the TDB, is 9 or 10.

Related information

- z/OS Architecture Principle of Operations

__TM_is_failure_persistent

Purpose

Queries whether the transaction was aborted because of a persistent reason.

Prototypes

long __TM_is_failure_persistent (long const *result*);

Parameter

`result`
: 2-bit *cc* returned by the `tbegin` hardware instruction.

Return value

This function returns 1 if the transaction was aborted because of a persistent reason; the value of *result* is 3. Otherwise, the function returns 0.

Related information

- z/OS Architecture Principle of Operations

__TM_is_footprint_exceeded

Purpose

Queries whether the transaction was aborted because of exceeding the maximum number of cache lines.

Prototypes

long __TM_is_footprint_exceeded (void* const *TM_buff*);

Parameter

`TM_buff`
: The address of a 256-byte transaction diagnostic block (TDB) that contains diagnostic information.

Return value

This function returns 1 if both of the following qualifications are met; otherwise, it returns 0:

- The TDB is valid.
- The transaction was aborted because the maximum number of cache lines was exceeded. The transaction abort code, which is the second word of the TDB, is 7 or 8.

Related information

- z/OS Architecture Principle of Operations

__TM_is_illegal

Purpose

Queries whether the transaction was aborted because of an illegal attempt, such as an instruction not permitted in transactional mode or other kind of illegal access.

Prototypes

long __TM_is_illegal (void* const *TM_buff*);

Parameter

TM_buff
The address of a 256-byte transaction diagnostic block (TDB) that contains diagnostic information.

Return value

This function returns 1 if both of the following qualifications are met; otherwise, it returns 0:

- The TDB is valid.
- The transaction was aborted because of an illegal attempt. The transaction abort code, which is the second word of the TDB, is 11 or 4.

Related information

- z/OS Architecture Principle of Operations

__TM_is_named_user_abort

Purpose

Queries whether the transaction failed because of a user abort instruction and gets the transaction abort code.

Prototypes

long __TM_is_named_user_abort (void* const *TM_buff*, unsigned char* *code*);

Parameter

code
The address of the memory location to save the transaction abort code.

TM_buff
The address of a 256-byte transaction diagnostic block (TDB) that contains diagnostic information.

Return value

This function returns 1 if both of the following qualifications are met; otherwise, it returns 0:

- The TDB is valid.
- The transaction failed because of a user abort instruction. The transaction abort code, which is the second word of the TDB, is no less than 256.

When both of the preceding qualifications are met, *code* is set to the transaction abort code minus 256. The value of *code* is also passed to the `tabort` hardware

instruction. When either of the preceding qualifications is not met, *code* is set to 0.

Related information

- "__TM_is_user_abort"
- z/OS Architecture Principle of Operations

__TM_is_nested_too_deep

Purpose

Queries whether the transaction was aborted because of trying to exceed the maximum nesting depth.

Prototypes

long __TM_is_nested_too_deep (void* const *TM_buff*);

Parameter

TM_buff
: The address of a 256-byte transaction diagnostic block (TDB) that contains diagnostic information.

Return value

This function returns 1 if both of the following qualifications are met; otherwise, it returns 0:

- The TDB is valid.
- The transaction was aborted because of trying to exceed the maximum nesting depth. The transaction abort code, which is the second word of the TDB, is 13.

Related information

- z/OS Architecture Principle of Operations

__TM_is_user_abort

Purpose

Queries whether the transaction failed because of a user abort instruction.

Prototypes

long __TM_is_user_abort (void* const *TM_buff*);

Parameter

TM_buff
: The address of a 256-byte transaction diagnostic block (TDB) that contains diagnostic information.

Return value

This function returns 1 if both of the following qualifications are met; otherwise, it returns 0:

- The TDB is valid.
- The transaction failed because of a user abort instruction. The transaction abort code, which is the second word of the TDB, is no less than 256.

Related information

- "__TM_is_named_user_abort" on page 209
- z/OS Architecture Principle of Operations

__TM_nesting_depth

Purpose

Returns the current nesting depth. If the thread is not in the transactional state, the function returns the depth at which the most recent transaction was aborted.

Prototypes

long __TM_nesting_depth (void* const *TM_buff*);

Parameter

TM_buff
: The address of a 256-byte transaction diagnostic block (TDB) that contains diagnostic information.

Return value

If the thread is in the transactional state, this function returns the current nesting depth. Otherwise, the function returns the depth at which the most recent transaction was aborted. The function returns 0 if the transaction is completed successfully.

The compiler uses ETNDG to get the current nesting depth. If the result is 0, the nesting depth is obtained from byte 6 and byte 7 of the TDB.

Related information

- z/OS Architecture Principle of Operations

Transaction store functions

__TM_non_transactional_store

Purpose

Stores a value to the specified memory location.

Prototypes

void __TM_non_transactional_store (void* const *addr*, long long const *value*);

Parameter

addr
: The address of the memory location to store the specified value.

value
: The value to be stored.

Usage

This function indicates that 8 bytes provided by *value* are stored at the address pointed by *addr*. The store is non-transactional.

Related information

- z/OS Architecture Principle of Operations

Notices

Programming interfaces: Intended programming interfaces allow the customer to write programs to obtain the services of IBM XL C/C++ for Linux on z Systems.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those

© Copyright IBM Corp. 2015

websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who want to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Intellectual Property Dept. for Rational Software
IBM Corporation
5 Technology Park Drive
Westford, MA 01886
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating

platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 2015.

PRIVACY POLICY CONSIDERATIONS:

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, or to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details in the section entitled "Cookies, Web Beacons and Other Technologies," and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

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" at http://www.ibm.com/legal/copytrade.shtml.

Adobe is a registered trademark of Adobe Systems Incorporated in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Index

Special characters

--help compiler option 37
--version (-qversion) compiler option 38
-qdbgfmt compiler option 83
-qhelp compiler option 37
-qreport compiler option 130
-qsaveopt compiler option 132

A

alias 72
  -qalias compiler option 72
  pragma disjoint 161
alignment 70
  -fpack-struct (-qalign) compiler option 70
  pragma align 70
  pragma pack 166
appending macro definitions, preprocessed output 60
architecture 7, 92
  -m64 compiler option 96
  -mtune compiler option 94
  -qarch compiler option 92
  -qtune compiler option 94
  architecture combination 95
  macros 175
arrays
  padding 104

B

basic example, described ix
built-in functions 179
  block-related 181
  cache-related 180
  fixed-point 179
  GCC atomic memory access 181
  hardware 191
  miscellaneous 190
  transactional memory 203

C

cleanpdf command 126
compatibility
  compatibility
    options for compatibility 34
compiler options 5
  architecture-specific 7
  performance optimization 31
  resolving conflicts 6
  specifying compiler options 5
    command line 5
    configuration file 5
    source files 6
  summary of command line options 23
compiler predefined macros 171
configuration 19
  custom configuration files 19
  specifying compiler options 5
configuration file 46

D

data types 96
  -mzvector compiler option 96

E

environment variables
  compile-time and link-time 18
  setting 17
error checking and debugging 28
  -g compiler option 80
  -qlinedebug compiler option 118

G

GCC options 153

H

high order transformation 104

I

implicit timestamps 138
inlining 65
interprocedural analysis (IPA) 109
invocations 1
  compiler or components 1
  preprocessor 8
  selecting 1
  syntax 2

L

language level 144
language standards 144
lib*.a library files 91
lib*.so library files 91
libraries
  redistributable 12
  XL C/C++ 12
linker 10
  invoking 10
linking 10
  options that control linking 33
  order of linking 11
listing 13
  -qlist compiler option 119
  options that control listings and messages 31
loop optimization 164

M

macro definitions, preprocessed output 60
macros
  related to architecture 175
  related to compiler options 174
  related to language features 176
  related to the compiler 172
  related to the platform 173
mergepdf 126

O

optimization 31
  -O compiler option 50
  -qalias compiler option 72
  -qoptimize compiler option 50
  controlling, using option_override pragma 164
  loop optimization 31
    -qhot compiler option 104
  options for performance optimization 31

P

performance 31
  -O compiler option 50
  -qalias compiler option 72
  -qoptimize compiler option 50
pragmas 160
profile-directed feedback (PDF) 123
  -qpdf1 compiler option 123
  -qpdf2 compiler option 123
profiling 98
  -qpdf1 compiler option 123
  -qpdf2 compiler option 123
  -qshowpdf compiler option 134

S

shared objects 141
  -shared (-qmkshrobj) 141
showpdf 126

T

target machine 92
templates
  -qtmplinst compiler option 139
tuning 94
  -march compiler option 94
  -mtune compiler option 94
  -qarch compiler option 94
  -qtune compiler option 94

© Copyright IBM Corp. 2015

V

vector data types 96
-mzvector compiler option 96
vector processing 78
-mzvector compiler option 96
virtual function table (VFT) 65
-fdump-class-hierarchy (-qdump_class_hierarchy) 65

IBM®

Product Number: 5725-N01

Printed in USA

SC27-5998-01