



VisualAge C++ for AIX Compiler Reference



VisualAge C++ for AIX Compiler Reference

Before using this information and the product it supports, be sure to read the information in "Notices" on page 411.

May 2002 Edition

This edition applies to Version 6 Release 0 of VisualAge C++ Professional for AIX (product number 5765-F56) and to all subsequent releases and modifications until otherwise indicated in new editions.

IBM® welcomes your comments. You can send them by either of the following methods:

- Internet: compinfo@ca.ibm.com

Be sure to include your e-mail address if you want a reply.

- By mail to the following address:

IBM Canada Ltd. Laboratory
Information Development
B3/KB7/8200/MKM
8200 Warden Avenue
Markham, Ontario, Canada L6G 1C7

Include the title and order number of this book, and the page number or topic related to your comment.

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

© Copyright International Business Machines Corporation 1995,2002. All rights reserved.

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

Contents

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

Part 1. Concepts 1

VisualAge C++ Compiler	3
Compiler Modes	3
Object Models	4
Compiler Options.	5
Types of Input Files	6
Types of Output Files	7
Compiler Message and Listing Information	8
Compiler Messages	8
Compiler Listings.	8

Program Parallelization	9
IBM SMP Directives	9
OpenMP Directives.	10
Countable Loops	11
Reduction Operations in Parallelized Loops	12
Shared and Private Variables in a Parallel Environment	13

Using VisualAge C++ with Other Programming Languages.	15
----------------------------------------------------------------------	-----------

Part 2. Tasks 17

Set Up the Compilation Environment	19
Set Environment Variables	19
Set Environment Variables in bsh, ksh, or sh Shells	19
Set Environment Variables in csh Shell	19
Set Environment Variables to Select 64- or 32-bit Modes	20
Set Parallel Processing Run-time Options	20
Set Environment Variables for the Message and Help Files	20

Invoke the Compiler	23
Invoke the Linkage Editor	23

Specify Compiler Options	25
Specify Compiler Options on the Command Line.	25
-q Options.	25
Flag Options	26
Specify Compiler Options in Your Program Source Files	27
Specify Compiler Options in a Configuration File.	27
Tailor a Configuration File	28
Configuration File Attributes	28

Specify Compiler Options for Architecture-Specific, 32- or 64-bit Compilation	29
Resolving Conflicting Compiler Options.	31

Specify Path Names for Include Files	33
Directory Search Sequence for Include Files Using Relative Path Names	33

Structure a Program that Uses Templates	35
Declaration of Stack in stack.h	35
Declaration of operator Functions in stack.c	35
Template Functions Declared Inline and Template Functions With Internal Linkage	36
Template Functions Defined within the Compilation Unit	37
Use -qtempinc to Generate Template Functions Automatically	38
How the Compiler Generates the Function Definitions.	39
Specifying the Template-Implementation File	40
Specifying a Different Path for the tempinc Subdirectory	40
Regenerating the Template Instantiation File	40
Breaking a Template Instantiation File into More Than One File	40
Contents of Template Instantiation File	40
Using #pragma Directives in Header Files	41
Considerations for Shared Libraries	42
Use -qnotempinc to Define Template Functions	42
Use -qtemplateregistry to Define Template Functions	43
Recompiling Parts of Your Program After Making Source Changes	43

Control Parallel Processing with Pragmas	45
-----------------------------------------------------------	-----------

Use C and C++ with Other Programming Languages.	47
Interlanguage Calling Conventions	47
Corresponding Data Types	47
Special Treatment of Character and Aggregate Data.	48
Use the Subroutine Linkage Conventions in Interlanguage Calls.	49
Interlanguage Calls - Parameter Passing.	50
Interlanguage Calls - Call by Reference Parameters	51
Interlanguage Calls - Call by Value Parameters	52
Interlanguage Calls - Rules for Passing Parameters by Value	52
Interlanguage Calls - Pointers to Functions	54
Interlanguage Calls - Function Return Values	54
Interlanguage Calls - Stack Floor	55

Interlanguage Calls - Stack Overflow	55
Interlanguage Calls - Traceback Table.	56
Interlanguage Calls - Type Encoding and Checking	56
Sample Program: C Calling Fortran	57

Part 3. Reference 59

Compiler Options 61

Compiler Command Line Options.	61
+ (plus sign)	71
# (pound sign)	72
32, 64	73
aggrcopy	74
alias	75
align.	77
alloca	81
ansialias	82
arch	83
assert	86
attr	87
B	88
b	89
bitfields	90
bmaxdata	91
brtl	92
C	93
c	94
cache	95
chars	97
check	98
cinc	100
compact	101
cpluscmt	102
D	106
dataimported	108
datalocal	109
dbxextra	110
digraph	111
dollar	113
dpcl	114
E	115
e	117
eh	118
enum	119
expfile.	125
extchk	126
F	127
f.	128
fdpr	129
flag.	130
float	131
flttrap	135
fold	137
fullpath	138
funcsect	139
G	140
g	141
genproto	142
halt.	143

haltonmsg	144
heapdebug	145
hot	146
hsflt	148
hssngl.	149
I.	150
idirfirst	151
ignerrno	152
ignprag	153
info	154
initauto	157
inlgue	158
inline	159
ipa	163
isolated_call	170
keepinlines	171
keyword	172
L	173
l.	174
langlvl.	175
largepage.	189
ldbl128, longdouble	190
libansi.	191
linedebug	192
list	193
listopt	194
longlit	195
longlong	197
M	198
ma	199
macpstr	200
maf.	203
makedep	204
maxerr	206
maxmem	208
mbcs, dbcs	209
mkshrobj	210
namemangling	214
O, optimize	215
o	219
objmodel	220
oldpassbyvalue.	221
P	222
p	223
pascal	224
path	225
pdf1, pdf2	226
pg	229
phsinfo	230
print	231
priority	232
proclocal, procimported, procunknown.	233
proto	235
Q	236
r.	239
report	240
rndflt	241
rndsngl	243
ro	244
roconst	245
rrm.	246

rtti	247	#pragma langlvl	317
S	248	#pragma leaves.	318
s	249	#pragma map	319
showinc	250	#pragma mc_func	321
smallstack	251	#pragma namemangling.	322
smp	252	#pragma nameManglingRule	323
source	254	#pragma object_model	324
spill	255	#pragma options	325
spnans	256	#pragma option_override	331
srcmsg	257	#pragma pack	332
staticinline	258	#pragma pass_by_value	335
statsym	259	#pragma priority	336
stdinc	260	#pragma reachable	337
strict	261	#pragma reg_killed_by	338
strict_induction.	262	#pragma report.	339
suppress	263	#pragma strings	341
symtab	264	#pragma unroll.	342
syntaxonly	265	Pragmas to Control Parallel Processing	344
t	266	#pragma ibm critical	346
tabsize.	267	#pragma ibm independent_calls	347
tbtable.	268	#pragma ibm independent_loop	348
tempinc	269	#pragma ibm iterations	349
templaterecompile.	270	#pragma ibm parallel_loop	350
templatereregistry	271	#pragma ibm permutation	351
tempmax	272	#pragma ibm schedule	352
threaded	273	#pragma ibm sequential_loop	354
tmplparse	274	#pragma omp atomic.	355
tocdata	275	#pragma omp parallel	356
tocmerge	276	#pragma omp for	358
tune	277	#pragma omp ordered	362
twolink	279	#pragma omp parallel for	363
U	281	#pragma omp section, #pragma omp sections	364
unique	282	#pragma omp parallel sections	366
unroll	283	#pragma omp single	367
unwind	285	#pragma omp master.	368
upconv	286	#pragma omp critical.	369
V	287	#pragma omp barrier.	370
v	288	#pragma omp flush	371
vftable.	289	#pragma omp threadprivate	372
W	290	Acceptable Compiler Mode and Processor	
w	291	Architecture Combinations	373
warn64	292	Compiler Messages.	379
xcall	293	Message Severity Levels and Compiler Response	379
xref.	294	Compiler Return Codes	379
y	295	Compiler Message Format	380
Z	296	Parallel Processing Support	383
General Purpose Pragmas	297	IBM SMP Run-time Options for Parallel Processing	383
#pragma align	299	Scheduling Algorithm Options	383
#pragma alloca	300	Parallel Environment Options	384
#pragma chars	301	Performance Tuning Options	384
#pragma comment.	302	Dynamic Profiling Options	385
#pragma define.	303	OpenMP Run-time Options for Parallel Processing	386
#pragma disjoint	304	Scheduling Algorithm Environment Variable	386
#pragma enum	305	Parallel Environment Environment Variables	387
#pragma execution_frequency	306	Dynamic Profiling Environment Variable . . .	387
#pragma hashome.	308	Built-in Functions Used for Parallel Processing	388
#pragma ibm snapshot	309	Part 4. Appendixes	391
#pragma implementation	310		
#pragma info	311		
#pragma ishome	314		
#pragma isolated_call	315		

Appendix A. Built-in Functions 393

General Purpose Built-in Functions 393
LIBANSI Built-in Functions. 394
Built-in Functions for PowerPC Processors 394

**Appendix B. National Languages
Support in VisualAge C++ 401**

Converting Files Containing Multibyte Data to
New Code Pages 401
Multibyte Character Support 401
 String Literals and Character Constants 401
 Preprocessor Directives 402
 Macro Definitions 402
 Compiler Options 402
 File Names and Comments. 403

Restrictions 403

Appendix C. Problem Solving 405

Message Catalog Errors 405
Correcting Paging Space Errors During
Compilation. 405

Appendix D. ASCII Character Set . . . 407

Notices 411

Programming Interface Information 413
Trademarks and Service Marks 413
Industry Standards 413

How to read syntax diagrams

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

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

Symbols

The following symbols may be displayed in syntax diagrams:

Symbol	Definition
▶—	Indicates the beginning of the syntax diagram.
—→	Indicates that the syntax diagram is continued to the next line.
▶—	Indicates that the syntax is continued from the previous line.
—▶	Indicates the end of the syntax diagram.

Syntax items

Syntax diagrams contain many different items. Syntax items include:

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

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

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

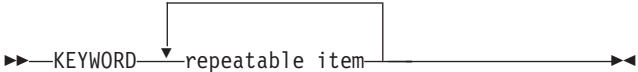

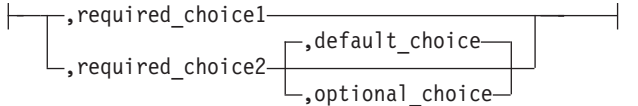
Syntax examples

The following table provides syntax examples.

Table 1. Syntax examples

Item	Syntax example
Required item.	►►—KEYWORD—required_item—◄◄
Required items appear on the main path of the horizontal line. You must specify these items.	
Required choice.	►►—KEYWORD— ┌required_choice1 └required_choice2—◄◄
A required choice (two or more items) appears in a vertical stack on the main path of the horizontal line. You must choose one of the items in the stack.	
Optional item.	►►—KEYWORD— ┌optional_item—◄◄
Optional items appear below the main path of the horizontal line.	
Optional choice.	►►—KEYWORD— ┌optional_choice1 └optional_choice2—◄◄
A optional choice (two or more items) appear in a vertical stack below the main path of the horizontal line. You may choose one of the items in the stack.	
Default.	►►—KEYWORD— ┌default_choice1 ┌optional_choice2 └optional_choice3—◄◄
Default items appear above the main path of the horizontal line. The remaining items (required or optional) appear on (required) or below (optional) the main path of the horizontal line. The following example displays a default with optional items.	
Variable.	►►—KEYWORD— <i>variable</i> —◄◄
Variables appear in lowercase italics. They represent names or values.	

Table 1. Syntax examples (continued)

Item	Syntax example
Repeatable item.	
<p>An arrow returning to the left above the main path of the horizontal line indicates an item that can be repeated.</p> <p>An arrow returning to the left above a group of repeatable items indicates that one of the items can be selected, or a single item can be repeated.</p>	
Fragment.	<p>fragment:</p> 

Part 1. Concepts

VisualAge C++ Compiler

You can use IBM VisualAge C++ as a C compiler for files with a `.c` (small c) suffix, or as a C++ compiler for files with a `.C` (capital C), `.cc`, `.cpp`, or `.cxx` suffix. The compiler processes your program source files to create an executable object module.

Note: Throughout these pages, the `x1C` command invocation is used to describe the actions of the compiler. You can, however, substitute other forms of the compiler invocation command if your particular environment requires it, and compiler option usage will remain the same unless otherwise specified.

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

- “Compiler Modes”
- “Object Models” on page 4
- “Compiler Options” on page 5
- “Types of Input Files” on page 6
- “Types of Output Files” on page 7
- “Compiler Message and Listing Information” on page 8

Compiler Modes

Several forms of VisualAge C++ compiler invocation commands support various version levels of the C and C++ languages. In most cases, you should use the `x1C` command to compile your C++ source files, and the `x1c` command to compile C source files. Use the `x1C` command when you have both C and C++ source files..

You can, however, use other forms of the command if your particular environment and file systems require it. The various compiler invocation commands are:

x1C	x1C128	x1C_r	x1C128_r	x1C_r4	x1C128_r4	x1C_r7	x1C128_r7
x1c	x1c128	x1c_r	x1c_r4	x1c_r7			
cc	cc128	cc_r	cc_r4	cc_r7			
c89							

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

- | | |
|------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| x1C | Invokes the compiler so that source files are compiled as C++ language source code.

Files with <code>.c</code> suffixes, assuming you have not used the <code>++</code> compiler option, are compiled as C language source code with a default language level of <code>ansi</code> , and compiler option <code>-qansialias</code> to allow type-based aliasing.

If any of your source files are C++, you must use this invocation to link with the correct runtime libraries. |
| x1c | Invokes the compiler for C source files with a default language level of <code>ansi</code> , and compiler option <code>-qansialias</code> to allow type-based aliasing. |

- cc** Invokes the compiler for C source files with a default language level of **extended** and compiler options **-qnor** and **-qnorconst** (to provide compatibility with the RT compiler and placement of string literals or constant values in read/write storage). Use this invocation for legacy C code that does not require compliance with ANSI C.
- c89** Invokes the compiler for C source files, with a default language level of **ansi**, and specifies compiler options **-qansialias** (to allow type based aliasing) and **-qnolonglong** (disabling use of **long long**), and sets **-D_ANSI_C_SOURCE** (for ANSI-conformant headers). Use this invocation for strict conformance to the ANSI standard (ISO/IEC 9899:1990).

IBM VisualAge C++ provides variations on the four basic compiler invocations. These variations are described below:

- 128-suffixed Invocations** All **128-suffixed** invocation commands are functionally similar to their corresponding base compiler invocations. They specify the **-qldb128** option, which increases the length of **long double** types in your program from 64 to 128 bits. They also link with the 128 versions of the C and C++ runtimes.
- _r-suffixed Invocations** All **_r-suffixed** invocations additionally set the macro names **-D_THREAD_SAFE** and add the libraries **-L/usr/lib/threads**, **-lc** and **-lpthreads**. The compiler option **-qthreaded** is also added. Use these commands if you want to create Posix threaded applications.

AIX 4.1 and 4.2 support Posix Draft 7. AIX 4.3 supports Draft 10. The **_r7** invocations are provided on AIX 4.3 to help with migration to Draft 10. See **-qthreaded** for additional information. The **_r4** invocations should be used for DCE threaded applications.

Migrating AIX Version 3.2.5 DCE Applications to AIX Version 4.3.3 and higher

The main invocation commands (except **c89**) have additional **_r4-suffixed** forms. These forms provide compatibility between DCE applications written for AIX Version 3.2.5 and AIX Version 4. They link your application to the correct AIX Version 4 DCE libraries, providing compatibility between the latest version of the pthreads library and the earlier versions supported on AIX Version 3.2.5.

Related Tasks

"Invoke the Compiler" on page 23

Related References

"Compiler Command Line Options" on page 61

"General Purpose Pragmas" on page 297

"Pragmas to Control Parallel Processing" on page 344

"threaded" on page 273

Object Models

VisualAge C++ lets you compile your program to either of two object models. The two object models are:

- **compat**
- **ibm**

The two object models differ in the following areas:

- Layout for the virtual function table
- Virtual base class support

- Name mangling scheme

Select `compat` if you need your runtime module to be compatible with any runtime modules compiled with the `compat` object module or with previous versions of VisualAge C++.

Select `ibm` if you want improved performance. This is especially true for class hierarchies with many virtual base classes. The size of the derived class is considerably smaller and access to the virtual function table is faster.

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

Use the `-qobjmodel` compiler option or the `object_model` pragma to specify a target object model.

Related References

“`objmodel`” on page 220

“`#pragma object_model`” on page 324

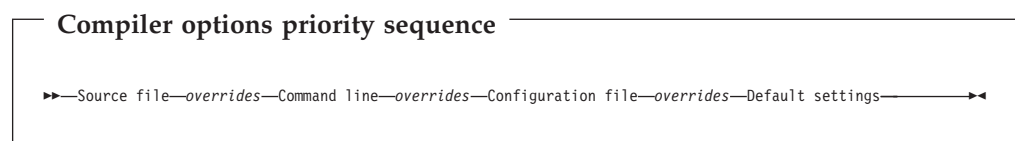
Compiler Options

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

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

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

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



Option conflicts that do not follow this priority sequence are described in “Resolving Conflicting Compiler Options” on page 31.

Related Tasks

“Invoke the Compiler” on page 23

“Specify Compiler Options” on page 25

“Resolving Conflicting Compiler Options” on page 31

Related References

“Compiler Command Line Options” on page 61

“General Purpose Pragas” on page 297

“Pragas to Control Parallel Processing” on page 344

Types of Input Files

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

C and C++ Source Files

These are files containing a C or C++ source module. To use the compiler as a C language compiler to compile a C language source file, the source file must have a `.c` (lowercase c) suffix, for example, `mysource.c`.

To use the compiler as a C++ language compiler, source file must have a `.C` (uppercase C), `.cc`, `.cpp`, or `.cxx` suffix. To compile other files as C++ source files, use the `-+` compiler option. All files specified with this option with a suffix other than `.o`, `.a`, or `.s`, are compiled as C++ source files.

The compiler will also accept source files with the `.i` suffix. This extension designates preprocessed source files.

The compiler processes the source files in the order in which they appear. If the compiler cannot find a specified source file, it produces an error message and the compiler proceeds to the next specified file. However, the link editor will not be run and temporary object files will be removed.

Your program can consist of several source files. All of these source files can be compiled at once using only one invocation of `x1C`. Although more than one source file can be compiled using a single invocation of the compiler, you can specify only one set of compiler options on the command line per invocation. Each distinct set of command-line compiler options that you want to specify requires a separate invocation.

By default, the `x1C` command preprocesses and compiles all the specified source files. Although you will usually want to use this default, you can use the `x1C` command to preprocess the source file without compiling by specifying either the `-E` or the `-P` option. If you specify the `-P` option, a preprocessed source file, `file_name.i`, is created and processing ends.

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

Preprocessed Source Files

Preprocessed source files have a `.i` suffix, for example, `file_name.i`. The `x1C` command sends the preprocessed source file, `file_name.i`, to the compiler where it is preprocessed again in the same way as a `.c` file. Preprocessed files are useful for checking macros and preprocessor directives.

Object Files

Object files must have a `.o` suffix, for example, `year.o`. Object files, library files, and nonstripped executable files serve as input to the linkage editor. After compilation, the linkage editor links all of the specified object files to create an executable file.

Assembler Files

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

Nonstripped Executable Files

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

Related Concepts

“Types of Output Files” on page 7

Related References

See:

strip command in Commands Reference, Volume 5: s through u
Files Reference

Types of Output Files

You can specify the following types of output files when invoking the IBM VisualAge C++ compiler.

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

The format of the a.out file is described in the *AIX Version 4 Files Reference*.

Object Files Object files must have a .o suffix, for example, year.o, unless the **-ofilename** option is specified.

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

You can link-edit the object files later into a single executable file using the **xlc** command.

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

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

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

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

A preprocessed source file, *file_name.i*, is produced for each source file, *file_name.c*.

Listing Files Listing files have a .lst suffix, for example, form.lst.

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

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

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

Related Concepts

“Types of Input Files” on page 6

Compiler Message and Listing Information

When the compiler encounters a programming error while compiling a C or C++ source program, it issues a diagnostic message to the standard error device and to the listing file.

Compiler Messages

The compiler issues messages specific to the C or C++ language, and XL messages common to all XL compilers.

C If you specify the compiler option **-qsrcmsg** and the error is applicable to a particular line of code, the reconstructed source line or partial source line is included with the error message in the stderr file. A reconstructed source line is a preprocessed source line that has all the macros expanded.

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

You can control the diagnostic messages issued, according to their severity, using either the **-qflag** option or the **-w** option. To get additional informational messages about potential problems in your program, use the **-qinfo** option.

Compiler Listings

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

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

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

Related References

“Message Severity Levels and Compiler Response” on page 379

“Compiler Message Format” on page 380

“flag” on page 130

“info” on page 154

“langlvl” on page 175

“source” on page 254

“srcmsg” on page 257

Program Parallelization

The compiler offers you three methods of implementing shared memory program parallelization. These are:

- Automatic parallelization of program loops.
- Explicit parallelization of C program code using IBM SMP pragma directives.
- Explicit parallelization of C and C++ program code using pragma directives compliant to the **OpenMP Application Program Interface** specification.

All methods of program parallelization are enabled when the **-qsmp** compiler option is in effect without the **omp** suboption. You can enable strict OpenMP compliance with the **-qsmp=omp** compiler option, but doing so will disable automatic parallelization.

Parallel regions of program code are executed by multiple threads, possibly running on multiple processors. The number of threads created is determined by the run-time options and calls to library functions. Work is distributed among available threads according to the specified scheduling algorithm.

Note: The **-qsmp** option must only be used together with thread-safe compiler invocation modes.

For more information about parallel programming support offered by the VisualAge C++ compiler, see the following topics in this section:

- “IBM SMP Directives”
- “OpenMP Directives” on page 10
- “Countable Loops” on page 11
- “Reduction Operations in Parallelized Loops” on page 12
- “Shared and Private Variables in a Parallel Environment” on page 13

For complete information about the OpenMP Specification, see:
OpenMP Web site
OpenMP Specification.

IBM SMP Directives

C IBM SMP directives exploit shared memory parallelism through the parallelization of *countable loops*. A loop is considered to be *countable* if it has any of the forms described in (Countable Loops).

The compiler can automatically locate and where possible parallelize all countable loops in your program code. In general, a countable loop is automatically parallelized only if all of the follow conditions are met:

- the order in which loop iterations start or end does not affect the results of the program.
- the loop does not contain I/O operations.
- floating point reductions inside the loop are not affected by round-off error, unless the **-qnostrict** option is in effect.
- the **-qnostrict_induction** compiler option is in effect.

- the `-qsmp` compiler option is in effect without the `omp` suboption. The compiler must be invoked using a thread-safe compiler mode.

You can also explicitly instruct the compiler to parallelize selected countable loops.

The VisualAge C++ compiler provides pragma directives that you can use to improve on automatic parallelization performed by the compiler. Pragmas fall into two general categories.

1. The first category of pragmas lets you give the compiler information on the characteristics of a specific countable loop. The compiler uses this information to perform more efficient automatic parallelization of the loop.
2. The second category gives you explicit control over parallelization. Use these pragmas to force or suppress parallelization of a loop, apply specific parallelization algorithms to a loop, and synchronize access to shared variables using critical sections.

Related Concepts

“OpenMP Directives”

“Countable Loops” on page 11

“Reduction Operations in Parallelized Loops” on page 12

“Shared and Private Variables in a Parallel Environment” on page 13

Related Tasks

“Set Parallel Processing Run-time Options” on page 20

“Control Parallel Processing with Pragmas” on page 45

Related References

“smp” on page 252

“Pragmas to Control Parallel Processing” on page 344

“IBM SMP Run-time Options for Parallel Processing” on page 383

“OpenMP Run-time Options for Parallel Processing” on page 386


“Built-in Functions Used for Parallel Processing” on page 388

For complete information about the OpenMP Specification, see:

OpenMP Web site

OpenMP Specification.

OpenMP Directives

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

Pragmas fall into four general categories:

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

Related Concepts

“IBM SMP Directives” on page 9

“Countable Loops” on page 11

“Reduction Operations in Parallelized Loops” on page 12
“Shared and Private Variables in a Parallel Environment” on page 13

Related Tasks

“Set Parallel Processing Run-time Options” on page 20
“Control Parallel Processing with Pragmas” on page 45

Related References

“smp” on page 252
“Pragmas to Control Parallel Processing” on page 344
“IBM SMP Run-time Options for Parallel Processing” on page 383
“OpenMP Run-time Options for Parallel Processing” on page 386
“Built-in Functions Used for Parallel Processing” on page 388

See also Parallel Programming in General Programming Concepts: Writing and Debugging Programs.

For complete information about the OpenMP Specification, see:
OpenMP Web site
OpenMP Specification.

Countable Loops

A loop is considered to be *countable* if it has any of the forms shown below, and:

- there is no branching into or out of the loop.
- the *incr_expr* expression is not within a critical section.

The following are examples of countable loops.

```
for ([iv]; exit_cond; incr_expr)
    statement

for ([iv]; exit_cond; [expr]) {
    [declaration_list]
    [statement_list]
    incr_expr;
    [statement_list]
}

while (exit_cond) {
    [declaration_list]
    [statement_list]
    incr_expr;
    [statement_list]
}

do {
    [declaration_list]
    [statement_list]
    incr_expr;
    [statement_list]
} while (exit_cond)
```

The following definitions apply to the above examples:

```
exit_cond  takes   iv <= ub
           form:   iv <  ub
                  iv >= ub
                  iv >  ub
```

<i>incr_expr</i>	takes	<i>++iv</i>
	form:	<i>iv++</i> <i>--iv</i> <i>iv--</i> <i>iv += incr</i> <i>iv -= incr</i> <i>iv = iv + incr</i> <i>iv = incr + iv</i> <i>iv = iv - incr</i>

iv Iteration variable. The iteration variable is a signed integer that has either automatic or register storage class, does not have its address taken, and is not modified anywhere in the loop except in *incr_expr*.

incr Loop invariant signed integer expression. The value of the expression is known at run-time and is not 0. *incr* cannot reference extern or static variables, pointers or pointer expressions, function calls, or variables that have their address taken.

ub Loop invariant signed integer expression. *ub* cannot reference extern or static variables, pointers or pointer expressions, function calls, or variables that have their address taken.

Related Concepts

“IBM SMP Directives” on page 9
 “OpenMP Directives” on page 10
 “Reduction Operations in Parallelized Loops”
 “Shared and Private Variables in a Parallel Environment” on page 13

Related Tasks

“Set Parallel Processing Run-time Options” on page 20
 “Control Parallel Processing with Pragmas” on page 45

Related References

“smp” on page 252
 “Pragmas to Control Parallel Processing” on page 344
 “IBM SMP Run-time Options for Parallel Processing” on page 383
 “OpenMP Run-time Options for Parallel Processing” on page 386
 “Built-in Functions Used for Parallel Processing” on page 388

Reduction Operations in Parallelized Loops

The compiler can recognize and properly handle most reduction operations in a loop during both automatic and explicit parallelization. In particular, it can handle reduction statements that have either of the following forms:

```
var = var op expr;  
var assign_op expr;
```


where:

<i>var</i>	Is an identifier designating an automatic or register variable that does not have its address taken and is not referenced anywhere else in the loop, including all loops that are nested. For example, in the following code, only S in the nested loop is recognized as a reduction: <pre>int i,j, S=0; #pragma ibm parallel_loop for (i= 0 ;i < N; i++) { S = S+ i; #pragma ibm parallel_loop for (j=0;j< M; j++) { S = S + j; } }</pre>
<i>op</i>	Is one of the following operators: + - * ^ &
<i>assign_op</i>	Is one of the following operators: += -= *= ^= = &=
<i>expr</i>	Is any valid expression.

Recognized reductions are listed by the **-qinfo=reduction** option. When using IBM directives, use critical sections to synchronize access to all reduction variables not recognized by the compiler. OpenMP directives provide you with mechanisms to specify reduction variables explicitly.

Shared and Private Variables in a Parallel Environment

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

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

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

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

The following code segments show examples of these default rules:

```

int E1;                                /* shared static */
void main (argc,...) {                 /* argc is shared */
    int i;                              /* shared automatic */
    void *p = malloc(...);             /* memory allocated by malloc */
                                        /* is accessible by all threads */
                                        /* and cannot be privatized */

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

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

        #pragma omp parallel
        {
            = b                            /* b is shared here because it */
                                            /* is another parallel region */
        }
    }
}

int E2;                                /*shared static */
void foo (int x) {                     /* x is private for the parallel */
                                        /* region it was called from */

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

```

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

Some OpenMP preprocessor directives let you specify visibility context for selected data variables. For more information, see OpenMP directive descriptions or the OpenMP C and C++ Application Program Interface specification.

Related Concepts

- “IBM SMP Directives” on page 9
- “OpenMP Directives” on page 10
- “Countable Loops” on page 11
- “Reduction Operations in Parallelized Loops” on page 12

Related Tasks

- “Set Parallel Processing Run-time Options” on page 20
- “Control Parallel Processing with Pragmas” on page 45

Related References

- “smp” on page 252
- “Pragmas to Control Parallel Processing” on page 344
- “IBM SMP Run-time Options for Parallel Processing” on page 383
- “OpenMP Run-time Options for Parallel Processing” on page 386
- “Built-in Functions Used for Parallel Processing” on page 388

Using VisualAge C++ with Other Programming Languages

With VisualAge C++, you can call functions written in other XL languages from your C and C++ programs. Similarly, the other XL language programs can call functions written in C and C++. You should already be familiar with the syntax of the languages you are using.

See "Use C and C++ with Other Programming Languages" on page 47 for more information.

Part 2. Tasks

Set Up the Compilation Environment

Before you compile your C or C++ programs, you must set up the environment variables and the configuration file for your application. For more information, see the following topics in this section:

- “Set Environment Variables”
- “Set Environment Variables to Select 64- or 32-bit Modes” on page 20
- “Set Parallel Processing Run-time Options” on page 20
- “Set Environment Variables for the Message and Help Files” on page 20

Set Environment Variables

You use different commands to set the environment variables depending on whether you are using the Bourne shell (**bsh** or **sh**), Korn shell (**ksh**), or C shell (**csh**). To determine the current shell, use the **echo** command:

```
echo $SHELL
```

The Bourne-shell path is **/bin/bsh** or **/bin/sh**. The Korn shell path is **/bin/ksh**. The C-shell path is **/bin/csh**.

For more information about the **NLSPATH** and **LANG** environment variables, see *System User's Guide: Operating System and Devices*. The AIX international language facilities are described in the *AIX General Programming Concepts*.

Set Environment Variables in bsh, ksh, or sh Shells

The following statements show how you can set environment variables in the Bourne or Korn shells:

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

To set the variables so that all users have access to them, add the commands to the file **/etc/profile**. To set them for a specific user only, add the commands to the file **.profile** in the user's home directory. The environment variables are set each time the user logs in.

Set Environment Variables in csh Shell

The following statements show how you can set environment variables in the C shell:

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

In the C shell, you cannot set the environment variables so that all users have access to them. 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.

Related Tasks

- “Set Environment Variables to Select 64- or 32-bit Modes” on page 20
- “Set Parallel Processing Run-time Options” on page 20
- “Set Environment Variables for the Message and Help Files” on page 20

Related References

See also:

System User's Guide: Operating System and Devices

AIX 5L Version 5.1 General Programming Concepts: Writing and Debugging Programs

Set Environment Variables to Select 64- or 32-bit Modes

You can set the `OBJECT_MODE` environment variable to specify a default compilation mode. Permissible values for the `OBJECT_MODE` environment variable are:

<code>(unset)</code>	Compiler programs generate and/or use 32-bit objects.
<code>32</code>	Compiler programs generate and/or use 32-bit objects.
<code>64</code>	Compiler programs generate and/or use 64-bit objects.
<code>32_64</code>	Set the compiler programs to accept both 32- and 64-bit objects. The compiler never functions in this mode, and using this choice may generate an error message, depending on other compilation options set at compile-time.

Related Tasks

"Specify Compiler Options for Architecture-Specific, 32- or 64-bit Compilation" on page 29

"Set Environment Variables" on page 19

Set Parallel Processing Run-time Options

The `XLSMPOPTS` environment variable sets options for programs using loop parallelization. For example, to have a program run-time create 4 threads and use dynamic scheduling with chunk size of 5, you would set the `XLSMPOPTS` environment variable as shown below:

```
XLSMPOPTS=PARTHDS=4:SCHEDULE=DYNAMIC=5
```

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

Related Tasks

"Set Environment Variables" on page 19

Related References

"IBM SMP Run-time Options for Parallel Processing" on page 383

"OpenMP Run-time Options for Parallel Processing" on page 386

Set Environment Variables for the Message and Help Files

Before using the compiler, you must install the message catalogs and help files and set the following two environment variables:

<code>LANG</code>	Specifies the national language for message and help files.
<code>NLSPATH</code>	Specifies the path name of the message and help files.

The `LANG` environment variable can be set to any of the locales provided on the system. See the description of locales in *AIX General Programming Concepts for IBM RISC System/6000* for more information.

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

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

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

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

Related Tasks

“Set Environment Variables” on page 19

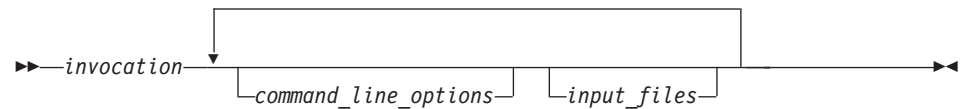
Related References

See also:

AIX 5L Version 5.1 General Programming Concepts: Writing and Debugging Programs

Invoke the Compiler

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



The parameters of the compiler invocation command can be the names of input files, compiler options, and linkage-editor options. Compiler options perform a wide variety of functions, such as setting compiler characteristics, describing the object code and compiler output to be produced, and performing some preprocessor functions.

To compile without link-editing, use the **-c** compiler option. The **-c** option stops the compiler after compilation is completed and produces as output, an object file *file_name.o* for each *file_name.c* input source file, unless the **-o** option was used to specify a different object filename. The linkage editor is not invoked. You can link-edit the object files later using the invocation command, specifying the object files without the **-c** option.

Notes:

1. Any object files produced from an earlier compilation are deleted as part of the compilation process, even if new object files are not produced.
2. By default, the invocation command calls *both* the compiler and the linkage editor. It passes linkage editor options to the linkage editor. Consequently, the invocation commands also accept all linkage editor options.

Invoke the Linkage Editor

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

Input Files

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

Library files are created by combining one or more files into a single archive file with the AIX **ar** command. For a description of the **ar** command, refer to the *AIX Version 4 Commands Reference*.

Output Files

The linkage editor generates an *executable file* and places it in your current directory. The default name for an executable file is **a.out**. To name the executable file explicitly, use the **-ofile_name** option with the **xlc** command,

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

You can invoke the linkage editor explicitly with the **ld** command. However, the compiler invocation commands set several linkage-editor options, and link some standard files into the executable output by default. In most cases, it is better to use one of the compiler invocation commands to link-edit your **.o** files.

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

Related Concepts

“Compiler Modes” on page 3

Related Tasks

“Specify Compiler Options on the Command Line” on page 25

Related References

“Compiler Command Line Options” on page 61

“Message Severity Levels and Compiler Response” on page 379

See also:

ld command in Commands Reference, Volume 5: s through u

Specify Compiler Options

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

- On the command line (see page 25)
- In your source program (see page 27)
- In a configuration (.cfg) file (see page 27)

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

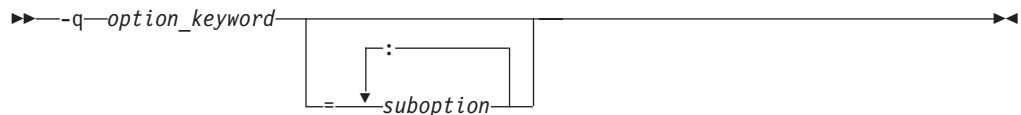
Specify Compiler Options on the Command Line

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

There are two kinds of command-line options:

- `-qoption_keyword` (compiler-specific)
- Flag options (available to compilers on AIX systems)

-q Options



Command-line options in the `-qoption_keyword` format are similar to on and off switches. For *most* `-q` options, if a given option is specified more than once, the last appearance of that option on the command line is the one recognized by the compiler. For example, `-qsource` turns on the source option to produce a compiler listing, and `-qnosource` turns off the source option so no source listing is produced. For example:

```
x1C -qnosource MyFirstProg.c -qsource MyNewProg.c
```

would produce a source listing for both `MyNewProg.c` and `MyFirstProg.c` because the last `source` option specified (`-qsource`) takes precedence.

You can have multiple `-qoption_keyword` instances in the same command line, but they must be separated by blanks. Option keywords can appear in either uppercase or lowercase, but you must specify the `-q` in lowercase. You can specify any `-qoption_keyword` before or after the file name. For example:

```
x1C -qLIST -qnomaf file.c  
x1C file.c -qxref -qsource
```

Some options have suboptions. You specify these with an equal sign following the `-qoption`. If the option permits more than one suboption, a colon (:) must separate each suboption from the next. For example:

```
x1C -qflag=w:e -qattr=full file.c
```

compiles the C source file `file.c` using the option **-qflag** to specify the severity level of messages to be reported, the suboptions `w` (warning) for the minimum level of severity to be reported on the listing, and `e` (error) for the minimum level of severity to be reported on the terminal. The option **-qattr** with suboption `full` will produce an attribute listing of all identifiers in the program.

Flag Options

The compilers available on AIX systems use a number of common conventional flag options. IBM VisualAge C++ supports these flags. Lowercase flags are different from their corresponding uppercase flags. For example, **-c** and **-C** are two different compiler options: **-c** specifies that the compiler should only preprocess and compile and not invoke the linkage editor, while **-C** can be used with **-P** or **-E** to specify that user comments should be preserved.

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

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

```
x1C stem.c -F/home/tools/test3/new.cfg:myc -qproclcal=sort:count
```

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

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

```
x1C -0cv file.c
```

has the same effect as:

```
x1C -0 -c -v file.c
```

and compiles the C source file `file.c` with optimization (**-O**) and reports on compiler progress (**-v**), but does not invoke the linkage editor (**-c**).

A flag option that takes arguments can be specified as part of a single string, but you can only use one flag that takes arguments, and it must be the last option specified. For example, you can use the **-o** flag (to specify a name for the executable file) together with other flags, only if the **-o** option and its argument are specified last. For example:

```
x1C -0votest test.c
```

has the same effect as:

```
x1C -0 -v -otest test.c
```

Most flag options are a single letter, but some are two letters. Note that **-pg** (extended profiling) is not the same as **-p -g** (profiling, **-p**, and generating debug information, **-g**). Take care not to specify two or more options in a single string if there is another option that uses that letter combination.

Related Concepts

“Compiler Options” on page 5

Related Tasks

“Invoke the Compiler” on page 23

“Specify Compiler Options in Your Program Source Files” on page 27

“Specify Compiler Options in a Configuration File”
“Resolving Conflicting Compiler Options” on page 31

Related References

“Compiler Command Line Options” on page 61

Specify Compiler Options in Your Program Source Files

You can specify compiler options within your program source by using `#pragma` directives.

A pragma is an implementation-defined instruction to the compiler. It has the general form given below, where *character_sequence* is a series of characters that giving a specific compiler instruction and arguments, if any.



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

Options specified with pragma directives in program source files override all other option settings.

These `#pragma` directives are listed in the detailed descriptions of the options to which they apply. For complete details on the various `#pragma` preprocessor directives, see *List of Pragma Preprocessor Directives*.

Related Concepts

“Compiler Options” on page 5

Related Tasks

“Invoke the Compiler” on page 23

“Specify Compiler Options on the Command Line” on page 25

“Specify Compiler Options in a Configuration File”

“Resolving Conflicting Compiler Options” on page 31

Related References

“General Purpose Pragmas” on page 297

Specify Compiler Options in a Configuration File

The default configuration file, (`/etc/vac.cfg`), specifies information that the compiler uses when you invoke it. This file defines values used by the compiler to compile C or C++ programs. You can make entries to this file to support specific compilation requirements or to support other C or C++ compilation environments.

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

Tailor a Configuration File

The default configuration file is installed to `/etc/vac.cfg`.

You can copy this file and make changes to the copy to support specific compilation requirements or to support other C or C++ compilation environments. To specify a configuration file other than the default, you use the `-F` option.

For example, to make `-qnor0` the default for the `x1C` compiler invocation command, add `-qnor0` to the `x1C` stanza in your copied version of the configuration file.

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

```
x1C myfile.c -Fmyconfig:SPECIAL
```

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

Configuration File Attributes

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

as	Path name to be used for the assembler. The default is <code>/bin/as</code> .
asopt	List of options for the assembler and not for the compiler. These override all normal processing by the compiler and are directed to the assembler specified in the as stanza. The string is formatted for the AIX <code>getopt()</code> subroutine as a concatenation of flag letters, with a letter followed by a colon (:) if the corresponding flag takes a parameter.
cppcode	Path name to be used for the code generation phase of the compiler. The default is <code>/usr/vacpp/exe/x1Ccode</code> .
ccomp	C Front end. The default is <code>/usr/vacpp/exe/x1Centry</code> .
codeopt	List of options for the code-generation phase of the compiler.
comp	C++ Front end. The default is <code>/usr/vacpp/exe/x1Centry</code> .
cppopt	List of options for the lexical analysis phase of the compiler.
crt	Path name of the object file passed as the first parameter to the linkage editor. If you do not specify either the <code>-p</code> or the <code>-pg</code> option, the crt value is used. The default is <code>/lib/crt0.o</code> .
csuffix	Suffix for source programs. The default is <code>c</code> (lowercase c).
dis	Path name of the disassembler. The default is <code>/usr/vacpp/exe/dis</code> .
gcr	Path name of the object file passed as the first parameter to the linkage editor. If you specify the <code>-pg</code> option, the gcr value is used. The default is <code>/lib/grt0.o</code> .
ld	Path name to be used to link C or C++ programs. The default is <code>/bin/ld</code> .
ldopt	List of options that are directed to the linkage editor part of the compiler. These override all normal processing by the compiler and are directed to the linkage editor. If the corresponding flag takes a parameter, the string is formatted for the AIX <code>getopt()</code> subroutine as a concatenation of flag letters, with a letter followed by a colon (:).
libraries2	Library options, separated by commas, that the compiler passes as the last parameters to the linkage editor. libraries2 specifies the libraries that the linkage editor is to use at link-edit time for both profiling and nonprofiling. The default is empty.
mcrt	Path name of the object file passed as the first parameter to the linkage editor if you have specified the <code>-p</code> option. The default is <code>/lib/mcrt0.o</code> .

options	A string of option flags, separated by commas, to be processed by the compiler as if they had been entered on the command line.
osuffix	The suffix for object files. The default is .o .
proflibs	Library options, separated by commas, that the compiler passes to the linkage editor when profiling options are specified. proflibs specifies the profiling libraries used by the linkage editor at link-edit time. The default is -L/lib/profiled and -L/usr/lib/profiled .
ssuffix	The suffix for assembler files. The default is .s .
use	Values for attributes are taken from the named stanza and from the local stanza. For single-valued attributes, values in the use stanza apply if no value is provided in the local, or default, stanza. For comma-separated lists, the values from the use stanza are added to the values from the local stanza.
xlC	The path name of the xlC compiler component. The default is /usr/vacpp/bin/xlC .

Related Concepts

“Compiler Options” on page 5

Related Tasks

“Invoke the Compiler” on page 23

“Specify Compiler Options on the Command Line” on page 25

“Specify Compiler Options in Your Program Source Files” on page 27

“Resolving Conflicting Compiler Options” on page 31

Related References

“Compiler Command Line Options” on page 61

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

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

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

1. Internal default (32-bit mode)
2. OBJECT_MODE environment variable setting, as follows:

OBJECT_MODE Setting	User-selected compilation-mode behavior, unless overridden by configuration file or command-line options
not set	32-bit compiler mode.
32	32-bit compiler mode.
64	64-bit compiler mode.
32_64	Fatal error and stop with following message, 1501-054 OBJECT_MODE=32_64 is not a valid setting for the compiler unless an explicit configuration file or command-line compiler-mode setting exists.
any other	Fatal error and stop with following message, 1501-055 OBJECT_MODE setting is not recognized and is not a valid setting for the compiler unless an explicit configuration file or command-line compiler-mode setting exists.

3. Configuration file settings
4. Command line compiler options (**-q32**, **-q64**, **-qarch**, **-qtune**)
5. Source file statements (**#pragma options tune=suboption**)

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

- *Compiler mode* is set according to the last-found instance of the **-q32** or **-q64** compiler options. If neither of these compiler options is chosen, the compiler mode is set by the value of the OBJECT_MODE environment variable.
- *Architecture target* is set according to the last-found instance of the **-qarch** compiler option, provided that the specified **-qarch** setting is compatible with the *compiler mode* setting. If the **-qarch** option is not set, the compiler assumes a **-qarch** setting of **com**.
- Tuning of the architecture target is set according to the last-found instance of the **-qtune** compiler option, provided that the **-qtune** setting is compatible with the *architecture target* and *compiler mode* settings. If the **-qtune** option is not set, the compiler assumes a default **-qtune** setting according to the **-qarch** setting in use.

Allowable combinations of these options are found in the **Acceptable Compiler Mode and Processor Architecture Combinations** table.

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

- **-q32** or **-q64** setting is incompatible with user-selected **-qarch** option.
Resolution: **-q32** or **-q64** setting overrides **-qarch** option; compiler issues a warning message, sets **-qarch** to **com**, and sets the **-qtune** option to the **-qarch** setting's default **-qtune** value.
- **-q32** or **-q64** setting is incompatible with user-selected **-qtune** option.
Resolution: **-q32** or **-q64** setting overrides **-qtune** option; compiler issues a warning message, and sets **-qtune** to the **-qarch** setting's default **-qtune** value.
- **-qarch** option is incompatible with user-selected **-qtune** option.
Resolution: Compiler issues a warning message, and sets **-qtune** to the **-qarch** setting's default **-qtune** value.
- Selected **-qarch** or **-qtune** options are not known to the compiler.
Resolution: Compiler issues a warning message, sets **-qarch** to **com**, and sets **-qtune** to the **-qarch** setting's default **-qtune** setting. The compiler mode (32- or 64-bit) is determined by the OBJECT_MODE environment variable or **-q32/-q64** compiler settings.

Related Concepts

"Compiler Options" on page 5

Related Tasks

"Invoke the Compiler" on page 23

"Specify Compiler Options on the Command Line" on page 25

"Set Environment Variables to Select 64- or 32-bit Modes" on page 20

Related References

"Compiler Command Line Options" on page 61

"Resolving Conflicting Compiler Options" on page 31

"Acceptable Compiler Mode and Processor Architecture Combinations" on page 373

Resolving Conflicting Compiler Options

In general, if more than one variation of the same option is specified (with the exception of **xref** and **attr**), the compiler uses the setting of the last one specified. Compiler options specified on the command line must appear in the order you want the compiler to process them.

If a command-line flag is valid for more than one compiler program (for example **-B**, **-W**, or **-I** applied to the compiler, linkage editor, and assembler program names), you must specify it in **cppopt**, **codeopt**, **inlineopt**, **ldopt**, or **asopt** in the configuration file. The command-line flags must appear in the order that they are to be directed to the appropriate compiler program.

Two exceptions to the rules of conflicting options are the **-Idirectory** and **-Ldirectory** options, which have cumulative effects when they are specified more than once.

In most cases, conflicting or incompatible options are resolved according to the precedence shown in the following figure:



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

Option	Conflicting Options	Resolution
-qhalt	Severity specified	Lowest severity specified.
-qnoprint	-qxref -qattr -qsource -qlistopt -qlist	-qnoprint
-qfloat=rsqrt	-qnoignerrno	Last option specified
-qxref	-qxref=FULL	-qxref=FULL
-qattr	-qattr=FULL	-qattr=FULL
-p -pg -qprofile	-p -pg -qprofile	Last option specified
-qhsflt	-qrndsngl -qspnans	-qhsflt
-qhssngl	-qrndsngl -qspnans	-qhssngl
-E	-P -o -S	-E
-P	-c -o -S	-P
-#	-v	-#
-F	-B -t -W -qpath configuration file settings	-B -t -W -qpath
-qpath	-B -t	-qpath overrides -B and -t
-S	-c	-S

Related Concepts

“Compiler Options” on page 5

Related Tasks

“Invoke the Compiler” on page 23

“Specify Compiler Options on the Command Line” on page 25

Related References

“Compiler Command Line Options” on page 61

Specify Path Names for Include Files

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

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

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

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

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

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

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

Directory Search Sequence for Include Files Using Relative Path Names

C and C++ define two versions of the **#include** preprocessor directive. IBM VisualAge C++ supports both. With the **#include** directive, you can search directories by enclosing the file name between < > or " " characters.

The result of using each method is as follows:

#include type	Directory Search Order
#include <file_name>	<ol style="list-style-type: none">1. If you specify the -Idirectory option, the compiler searches for <i>file_name</i> in the directory called <i>directory</i> first. If more than one directory is specified, the compiler searches the directories in the order that they appear on the command line.2. For C++ compilations, the compiler searches the directory /usr/vacpp/include.3. The compiler searches the directory /usr/include.
#include "file_name"	<ol style="list-style-type: none">1. Starts searching from the directory where your current source file resides. The current source file is the file that contains the directive #include "file_name".2. If you specify the option -Idirectory, the compiler searches for <i>file_name</i> in <i>directory</i>. If more than one directory is specified, the compiler searches the directories in the order that they appear on the command line.3. For C++ compilations, the compiler searches the directory /usr/vacpp/include.4. The compiler searches the directory /usr/include.

Notes:

1. *file_name* specifies the name of the file to be included, and can include a full or partial directory path to that file if you desire.

- If you specify a file name by itself, the compiler searches for the file in the directory search list.
 - If you specify a file name together with a partial directory path, the compiler appends the partial path to each directory in the search path, and tries to find the file in the completed directory path.
 - If you specify a full path name, the two versions of the **#include** directive have the same effect because the location of the file to be included is completely specified.
2. The only difference between the two versions of the **#include** directive is that the " " (user include) version first begins a search from the directory where your current source file resides. Typically, standard header files are included using the < > (system include) version, and header files that you create are included using the " " (user include) version.
 3. You can change the search order by specifying the **-qstdinc** and **-qidirfirst** options along with the **-Idirectory** option.

Use the **-qnostdinc** option to search only the directories specified with the **-Idirectory** option and the current source file directory, if applicable. For C programs, the **/usr/include** directory is not searched. For C++ programs, the **/usr/vacpp/include** and **/usr/include** directories are not searched.

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

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

Related References

"I" on page 150

Structure a Program that Uses Templates

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

See the following sections:

- “Declaration of `Stack` in `stack.h`”
- “Declaration of operator Functions in `stack.c`”
- “Template Functions Declared Inline and Template Functions With Internal Linkage” on page 36
- “Template Functions Defined within the Compilation Unit” on page 37
- “Use `-qtempinc` to Generate Template Functions Automatically” on page 38
- “Use `-qnotempinc` to Define Template Functions” on page 42
- “Use `-qtemplateregistry` to Define Template Functions” on page 43

Declaration of `Stack` in `stack.h`

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

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

Related Tasks

- “Structure a Program that Uses Templates”
- “Declaration of operator Functions in `stack.c`”
- “Template Functions Declared Inline and Template Functions With Internal Linkage” on page 36
- “Template Functions Defined within the Compilation Unit” on page 37
- “Use `-qtempinc` to Generate Template Functions Automatically” on page 38
- “Use `-qnotempinc` to Define Template Functions” on page 42
- “Use `-qtemplateregistry` to Define Template Functions” on page 43

Related References

- “`tempinc`” on page 269

Declaration of operator Functions in `stack.c`

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

```

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

```

Related Tasks

“Structure a Program that Uses Templates” on page 35

“Declaration of Stack in stack.h” on page 35

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

“Template Functions Defined within the Compilation Unit” on page 37

“Use `-qtempinc` to Generate Template Functions Automatically” on page 38

“Use `-qnotempinc` to Define Template Functions” on page 42

“Use `-qtemplateregistry` to Define Template Functions” on page 43

Related References

“tempinc” on page 269

Template Functions Declared Inline and Template Functions With Internal Linkage

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

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

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

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

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

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

Related Tasks

“Structure a Program that Uses Templates” on page 35

“Declaration of Stack in stack.h” on page 35

“Declaration of operator Functions in stack.c” on page 35
“Template Functions Defined within the Compilation Unit”
“Use -qtempinc to Generate Template Functions Automatically” on page 38
“Use -qnotempinc to Define Template Functions” on page 42
“Use -qtemplateregistry to Define Template Functions” on page 43

Related References

“tempinc” on page 269

Template Functions Defined within the Compilation Unit

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

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

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

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

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

will automatically generate code for these functions :

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

Related Tasks

“Structure a Program that Uses Templates” on page 35
“Declaration of Stack in stack.h” on page 35
“Declaration of operator Functions in stack.c” on page 35
“Template Functions Declared Inline and Template Functions With Internal Linkage” on page 36
“Use -qtempinc to Generate Template Functions Automatically” on page 38

“Use -qnotempinc to Define Template Functions” on page 42

“Use -qtemplateregistry to Define Template Functions” on page 43

Related References

“tempinc” on page 269

Use -qtempinc to Generate Template Functions Automatically

To avoid producing template code for the same function multiple times, use the compiler to automatically generate the template functions. This is the recommended way to use templates with the compiler.

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

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

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

To generate template functions automatically:

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

6. If you compile and then link at a different time, repeat any compiler options you specified at compile time, when you link. Using the same compiler options allows the compiler to properly compile the template-include files that are generated at compile time. For example, use the same path names for the `-Idirectory` option so that the compiler uses the same include files.

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

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

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

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

Note to Users of xLC V3.x: Previous versions of xLC instantiated every method of a class whether used or not. With VisualAge C++ V5.0 or later, xLC only instantiates the methods that are used. This may result in unresolved symbols for poorly organized code. Check your organization. Reference symbols in the source file where the symbols were generated with xLC V3.x.

How the Compiler Generates the Function Definitions

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

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

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

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

Specifying the Template-Implementation File

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

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

Specifying a Different Path for the tempinc Subdirectory

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

Regenerating the Template Instantiation File

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

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

If a compiler-generated file in a tempinc directory has compiler errors, the file will be recompiled whenever a link is done using the x1C compiler invocation with that tempinc directory. To avoid this recompilation, either fix the errors in the file or remove it from the tempinc directory.

Breaking a Template Instantiation File into More Than One File

Normally the compiler generates one template instantiation file for each template header file (either the default one or one specified by **#pragma implementation**). When compiled, the template instantiation file may be too large to be created by the compiler. If so, you can break the template instantiation file into more than one file using the **-qtempmax=number** compiler option. More than one object file will be created, all of them smaller in size than the first one.

Contents of Template Instantiation File

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

Example of a Typical Template-Include File

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

```

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

```

A descriptions of each line follows:

1. The header file that corresponds to the template-include file. The number in comments at the start of each #include line (in this case, /*0698421265*/) is a timestamp for the included file. The compiler uses this number to determine if the template-include file should be recompiled. A time stamp of zeroes (as in line 2.) means the compiler is to ignore the timestamp.
2. The template implementation file that corresponds to the header file in line 1.
3. Another header file that the compiler needs to compile the template-include file. All other header files that the compiler needs to compile the template-include file are inserted at this point. In this example, the type MyType is used as a template argument and was defined in the mytype.h header file (MyType is needed for the instantiation of template function foo).
4. Another include file inserted by the compiler. It is referenced in the function explicit instantiation at line 6 (ostream is needed for the instantiation of the operator <<).
5. Code for the member function foo of class template List is going to be generated, for the specific type MyType, by this explicit instantiation.
6. The operator << function has namespace scope. It is matching a template declaration in the file list.h and has its definition in list.c. The compiler inserted this explicit instantiation to force the generation of the function code.
7. count function is a function that has namespace scope. The compiler inserted this explicit instantiation to force the generation of the function code.

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

```

/*0709395703*/#include "/home/myapp/stack.h"
/*0000000000*/#include "/home/myapp/stack.c"
template int Stack<int,20>::operator<<(int);
template int Stack<int,20>::operator>>(int &);

```

Using #pragma Directives in Header Files

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

```

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

```

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

```

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

```

enum2 would be treated as small if **header2.h** followed **header1.h**. enum2 would be treated as int if **header2.h** preceded **header1.h** and if **header2.h** had no

`#pragma options enum=small` directive. In this example, you should specify `#pragma options enum=reset` at the end of `header1.h` to avoid any carry over to another file.

Considerations for Shared Libraries

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

If you use templates:

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

Related Tasks

“Structure a Program that Uses Templates” on page 35

“Declaration of Stack in `stack.h`” on page 35

“Declaration of operator Functions in `stack.c`” on page 35

“Template Functions Declared Inline and Template Functions With Internal Linkage” on page 36

“Template Functions Defined within the Compilation Unit” on page 37

“Use `-qnotempinc` to Define Template Functions”

“Use `-qtemplateregistry` to Define Template Functions” on page 43

Related References

“`tempinc`” on page 269

Use `-qnotempinc` to Define Template Functions

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

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

To structure your program without using automatic template generation:

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

In the Stack class template example above, the compiler generates the necessary function code if you include both `stack.h` and `stack.c` in all compilation units

which use instances of the Stack class. Code is generated for all the necessary functions. Code may be generated multiple times resulting in a very large object file.

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

Related Tasks

“Structure a Program that Uses Templates” on page 35

“Declaration of Stack in stack.h” on page 35

“Declaration of operator Functions in stack.c” on page 35

“Template Functions Declared Inline and Template Functions With Internal Linkage” on page 36

“Template Functions Defined within the Compilation Unit” on page 37

“Use `-qtempinc` to Generate Template Functions Automatically” on page 38

“Use `-qtemplateregistry` to Define Template Functions”

Related References

“tempinc” on page 269

Use `-qtemplateregistry` to Define Template Functions

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

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

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

Recompiling Parts of Your Program After Making Source Changes

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

For example, if you have compilation units A and B that both reference the same instantiation and you compile A first, then A’s object file will contain the code for the instantiation. If you modify A so that it no longer references the instantiation,

and you recompile A only, then its object file will no longer contain the code for the instantiation. Without further action an undefined symbol error will occur. To handle this situation the compiler will automatically recompile B using the same compiler options as it did for A, so that B's object file contains the code for the instantiation.

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

Related Tasks

"Structure a Program that Uses Templates" on page 35

"Declaration of Stack in stack.h" on page 35

"Declaration of operator Functions in stack.c" on page 35

"Template Functions Declared Inline and Template Functions With Internal Linkage" on page 36

"Template Functions Defined within the Compilation Unit" on page 37

"Use -qtempinc to Generate Template Functions Automatically" on page 38

"Use -qnotempinc to Define Template Functions" on page 42

Related References

"tempinc" on page 269

"templatercompile" on page 270

"templaterregistry" on page 271

Control Parallel Processing with Pragmas

Parallel processing operations are controlled by pragma directives in your program source. You can use either IBM SMP or OpenMP parallel processing directives. Each have their own usage characteristics.

IBM SMP Directives

► C

Syntax

```
#pragma ibm pragma_name_and_args  
<countable for|while|do loop>
```

Pragma directives must appear immediately before the section of code to which they apply. For most parallel processing pragma directives this section of code must be a countable loop, and the compiler will report an error if one is not found.

More than one parallel processing pragma directive can be applied to a countable loop. For example:

```
#pragma ibm independent_loop  
#pragma ibm independent_calls  
#pragma ibm schedule(static,5)  
<countable for|while|do loop>
```

Some pragma directives are mutually-exclusive of each other. If mutually-exclusive pragmas are specified for the same loop, the pragma last specified applies to the loop. In the example below, the **parallel_loop** pragma directive is applied to the loop, and the **sequential_loop** pragma directive is ignored.

```
#pragma ibm sequential_loop  
#pragma ibm parallel_loop
```

Other pragmas, if specified repeatedly for a given loop, have an additive effect. For example:

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

is equivalent to:

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

OpenMP Directives

► C ► C++

Syntax

```
#pragma omp pragma_name_and_args  
statement_block
```

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

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

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

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

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

Related Concepts

“Program Parallelization” on page 9

Related Tasks

“Set Parallel Processing Run-time Options” on page 20

Related References

“smp” on page 252

“Pragmas to Control Parallel Processing” on page 344

“IBM SMP Run-time Options for Parallel Processing” on page 383

“OpenMP Run-time Options for Parallel Processing” on page 386

“Built-in Functions Used for Parallel Processing” on page 388

For complete information about the OpenMP Specification, see:

OpenMP Web site

OpenMP Specification.

Use C and C++ with Other Programming Languages

You can use objects created in other programming languages in your C or C++ programs. The following topics in this section give an overview of programming considerations to follow when doing so.

- “Interlanguage Calling Conventions”
- “Corresponding Data Types”
- “Use the Subroutine Linkage Conventions in Interlanguage Calls” on page 49
- “Sample Program: C Calling Fortran” on page 57

Interlanguage Calling Conventions

You should follow these recommendations when writing C and C++ code to call functions written in other languages:

- Avoid using uppercase letters in identifiers. Fortran and Pascal use only lowercase letters for all external names. Although both fold external identifiers to lowercase by default, the Fortran compiler can be set to distinguish external names by case.
- Avoid using the underscore (`_`) and dollar sign (`$`) as the first character in identifiers, to prevent conflict with the naming conventions for the C and C++ language libraries.
- Avoid using long identifier names. The maximum number of significant characters in identifiers is 250 characters.

Related Concepts

“Using VisualAge C++ with Other Programming Languages” on page 15

Related Tasks

“Corresponding Data Types”

“Use the Subroutine Linkage Conventions in Interlanguage Calls” on page 49

“Sample Program: C Calling Fortran” on page 57

Corresponding Data Types

The following table shows the correspondence between the data types available in C/C++, Fortran, and Pascal. Several data types in C have no equivalent representation in Pascal or Fortran. Do not use them when programming for interlanguage calls. Blank table cells indicate that no matching data type exists.

C and C++ Data Types	Fortran Data Types	Pascal Data Types
Correspondence of Data Types among C, C++, Fortran, and Pascal		
bool	LOGICAL*4	
char	CHARACTER	CHAR
signed char	INTEGER*1 BYTE	PACKED -128..127
unsigned char	LOGICAL*1	PACKED 0..255
signed short int	INTEGER*2	PACKED -32768..32767
unsigned short int	LOGICAL*2	PACKED 0..65535
signed long int	INTEGER*4	INTEGER

C and C++ Data Types	Fortran Data Types	Pascal Data Types
unsigned long int	LOGICAL*4	—
signed long long int	INTEGER*8	—
unsigned long long int	LOGICAL*8	—
float	REAL REAL*4	SHORTREAL
double	REAL*8 DOUBLE PRECISION	REAL
long double (default)	REAL*8 DOUBLE PRECISION	REAL
long double (with -qlongdouble or -qldb128)	REAL*16	—
structure of two floats	COMPLEX COMPLEX*4	RECORD of two SHORTREALS
structure of two doubles	COMPLEX*16 DOUBLE COMPLEX	RECORD of two REALS
struct of two long doubles (default)	COMPLEX*16	—
struct	—	RECORD (see notes below)
enumeration	INTEGER*4	Enumeration
char[n]	CHARACTER*n	PACKED ARRAY[1..n] OF CHAR
array pointer (*) to type	Dimensioned variable (transposed)	ARRAY
pointer (*) to function	Functional Parameter	Functional Parameter
structure (with -qalign=packed)	Sequence derived type	PACKED RECORD

Special Treatment of Character and Aggregate Data

Most numeric data types have counterparts across the three languages. Character and aggregate data types require special treatment:

- Because of padding and alignment differences, C structures do not exactly correspond to the Pascal **RECORD** data type.
- C character strings are delimited by a '\0' character. In Fortran, all character variables and expressions have a length that is determined at compile time. If Fortran passes a string argument to another routine, it adds a hidden argument giving the length to the end of the argument list. This length argument must be explicitly declared in C. The C code should not assume a null terminator; the supplied or declared length should always be used. Use the **strncat**, **strncpy**, and **strncpy** functions of the C runtime library. These functions are described in the *Technical Reference, Volumes 1 and 2: Base Operating System and Extensions*.
- Pascal's STRING data type corresponds to a C structure. For example.:

```
VAR s: STRING(10);
```

is equivalent to:

```
struct {
    int length;
    char str [10];
};
```

where length contains the actual length of STRING.

- The **-qmacpstr** option converts Pascal string literals into null-terminated strings, where the first byte contains the length of the string.
- C and Pascal store array elements in row-major order (array elements in the same row occupy adjacent memory locations). Fortran stores array elements in ascending storage units in column-major order (array elements in the same column occupy adjacent memory locations). The following example shows how a two-dimensional array declared by A[3][2] in C, A[1..3,1..2] in Pascal, and by A(3,2) in Fortran, is stored:

Storage of a Two-Dimensional Array			
Storage Unit	C and C++ Element Name	Pascal Element Name	Fortran Element Name
Lowest	A[0][0]	A[1,1]	A(1,1)
	A[0][1]	A[1,2]	A(2,1)
	A[1][0]	A[2,1]	A(3,1)
	A[1][1]	A[2,2]	A(1,2)
	A[2][0]	A[3,1]	A(2,2)
Highest	A[2][1]	A[3,2]	A(3,2)

- In general, for a multidimensional array, if you list the elements of the array in the order they are laid out in memory, a row-major array will be such that the rightmost index varies fastest, while a column-major array will be such that the leftmost index varies fastest.

Related Concepts

“Using VisualAge C++ with Other Programming Languages” on page 15

Related Tasks

“Interlanguage Calling Conventions” on page 47

“Use the Subroutine Linkage Conventions in Interlanguage Calls”

“Sample Program: C Calling Fortran” on page 57

Related References

See also:

AIX 5L Version 5.1 Technical Reference: Base Operating System and Extensions, Volume 1 (A-P)

AIX 5L Version 5.1 Technical Reference: Base Operating System and Extensions, Volume 2 (Q-Z)

Use the Subroutine Linkage Conventions in Interlanguage Calls

Subroutine linkage conventions describe the machine state at subroutine entry and exit. Routines that are compiled separately in the same or different languages are linked when the programs are linked, and run when called.

These linkage convention provide fast and efficient subroutine linkage between languages. They specify how parameters are passed taking full advantage of floating-point registers (FPRs) and general-purpose registers (GPRs), and minimize the saving and restoring of registers on subroutine entry and exit.

- “Interlanguage Calls - Parameter Passing” on page 50

- “Interlanguage Calls - Call by Reference Parameters” on page 51
- “Interlanguage Calls - Call by Value Parameters” on page 52
- “Interlanguage Calls - Rules for Passing Parameters by Value” on page 52
- “Interlanguage Calls - Pointers to Functions” on page 54
- “Interlanguage Calls - Function Return Values” on page 54
- “Interlanguage Calls - Stack Floor” on page 55
- “Interlanguage Calls - Stack Overflow” on page 55
- “Interlanguage Calls - Traceback Table” on page 56
- “Interlanguage Calls - Type Encoding and Checking” on page 56
- “Sample Program: C Calling Fortran” on page 57

Related Concepts

“Using VisualAge C++ with Other Programming Languages” on page 15

Related Tasks

“Interlanguage Calling Conventions” on page 47

“Corresponding Data Types” on page 47

Interlanguage Calls - Parameter Passing

The RISC System/6000 linkage convention specifies the methods for parameter passing and whether return values are to be in FPRs, GPRs, or both. The GPRs and FPRs available for argument passing are specified in two fixed lists: R3-R10 and FP1-FP13.

Prototyping affects how parameters are passed and whether widening occurs:

Nonprototyped functions

In nonprototyped functions in the C language, floating-point arguments are widened to **double** and integral types are widened to **int**.

Prototyped functions

No widening conversions occur except in arguments passed to an ellipsis function. Floating-point **double** arguments are only passed in FPRs. If an ellipsis is present in the prototype, floating-point **double** arguments are passed in both FPRs and GPRs.

When there are more argument words than available parameter GPRs and FPRs, the remaining words are passed in storage on the stack. The values in storage are the same as if they were in registers. Space for more than 8 words of arguments (float and nonfloat) must be reserved on the stack even if all the arguments were passed in registers.

The size of the parameter area is sufficient to contain all the arguments passed on any call statement from a procedure associated with the stack frame. Although not all the arguments for a particular call actually appear in storage, they can be regarded as forming a list in this area, each one occupying one or more words.

The methods of passing parameters are as follows:

- In C, all function arguments are passed by value, and the called function receives a copy of the value passed to it.
- In Fortran, by default, arguments are passed by reference, and the called function receives the address of the value passed to it. You can use the **%VAL**

Fortran built-in function to pass by value. Refer to the *AIX XL Fortran Compiler/6000 User's Guide* for more information about using %VAL and interlanguage calls.

- In Pascal, the function declaration determines whether a parameter is expected to be passed by value or by reference.

Related Concepts

"Using VisualAge C++ with Other Programming Languages" on page 15

Related Tasks

"Interlanguage Calling Conventions" on page 47

"Corresponding Data Types" on page 47

"Use the Subroutine Linkage Conventions in Interlanguage Calls" on page 49

"Sample Program: C Calling Fortran" on page 57

"Interlanguage Calls - Call by Reference Parameters"

"Interlanguage Calls - Call by Value Parameters" on page 52

"Interlanguage Calls - Rules for Passing Parameters by Value" on page 52

"Interlanguage Calls - Pointers to Functions" on page 54

"Interlanguage Calls - Function Return Values" on page 54

"Interlanguage Calls - Stack Floor" on page 55

"Interlanguage Calls - Stack Overflow" on page 55

"Interlanguage Calls - Traceback Table" on page 56

"Interlanguage Calls - Type Encoding and Checking" on page 56

Interlanguage Calls - Call by Reference Parameters

For call-by-reference (as in Fortran), the address of the parameter is passed in a register.

When passing parameters by reference, if you write C or C++ functions that:

- you want to call from a Fortran program, declare all parameters as pointers.
- calls a program written in Fortran, all arguments must be pointers or scalars with the address operator.
- you want to call from a Pascal program, declare as pointers all parameters that the Pascal program treats as reference parameters.
- calls a program written in Pascal, all arguments corresponding to reference parameters must be pointers.

Related Concepts

"Using VisualAge C++ with Other Programming Languages" on page 15

Related Tasks

"Interlanguage Calling Conventions" on page 47

"Corresponding Data Types" on page 47

"Use the Subroutine Linkage Conventions in Interlanguage Calls" on page 49

"Sample Program: C Calling Fortran" on page 57

"Interlanguage Calls - Parameter Passing" on page 50

"Interlanguage Calls - Call by Value Parameters" on page 52

"Interlanguage Calls - Rules for Passing Parameters by Value" on page 52

"Interlanguage Calls - Pointers to Functions" on page 54

"Interlanguage Calls - Function Return Values" on page 54

"Interlanguage Calls - Stack Floor" on page 55

"Interlanguage Calls - Stack Overflow" on page 55

"Interlanguage Calls - Traceback Table" on page 56

"Interlanguage Calls - Type Encoding and Checking" on page 56

Interlanguage Calls - Call by Value Parameters

In prototype functions with a variable number of arguments— specified with an ellipsis, as in *function(...)*— the compiler widens all floating-point arguments to double precision. Integral arguments (except for **long int**) are widened to **int**. Because of this widening, some data types cannot be passed between Pascal and C without explicit conversions, and Pascal routines cannot have value parameters of certain data types.

The following information refers to call by value, as in C. In the following list, arguments are classified as floating values or nonfloating values:

- Each nonfloating scalar argument requires 1 word and appears in that word exactly as it would appear in a GPR. It is right-justified, if language semantics specify, and is word aligned.
- Each float value occupies 1 word, float doubles occupy 2 successive words in the list, and long doubles occupy either 2 or 4 words, depending on the setting of the **-qldbl128/-qlongdouble** option.
- Structure values appear in successive words as they would anywhere in storage, satisfying all appropriate alignment requirements. Structures are aligned to a fullword and occupy $(\text{sizeof}(\text{struct } X) + (\text{wordsize} - 1)) / \text{wordsize}$ fullwords, with any padding at the end. A structure smaller than a word is left-justified within its word or register. Larger structures can occupy multiple registers and can be passed partly in storage and partly in registers.
- Other aggregate values are passed *val-by-ref*; that is, the compiler actually passes their addresses and arranges for a copy to be made in the invoked program.
- A function pointer is passed as a pointer to the routine's function descriptor. The first word contains the entry-point address. See *Interlanguage Calls - Pointers to Functions* for more information.

Related Concepts

"Using VisualAge C++ with Other Programming Languages" on page 15

Related Tasks

"Interlanguage Calling Conventions" on page 47

"Corresponding Data Types" on page 47

"Use the Subroutine Linkage Conventions in Interlanguage Calls" on page 49

"Sample Program: C Calling Fortran" on page 57

"Interlanguage Calls - Parameter Passing" on page 50

"Interlanguage Calls - Call by Reference Parameters" on page 51

"Interlanguage Calls - Rules for Passing Parameters by Value"

"Interlanguage Calls - Pointers to Functions" on page 54

"Interlanguage Calls - Function Return Values" on page 54

"Interlanguage Calls - Stack Floor" on page 55

"Interlanguage Calls - Stack Overflow" on page 55

"Interlanguage Calls - Traceback Table" on page 56

"Interlanguage Calls - Type Encoding and Checking" on page 56

Interlanguage Calls - Rules for Passing Parameters by Value

The following is a 32-bit example of a call to a prototyped function:

```
int i, j; //32 bits each
long k; //32 bits
double d1, d2;
float f1;
short int s1;
```

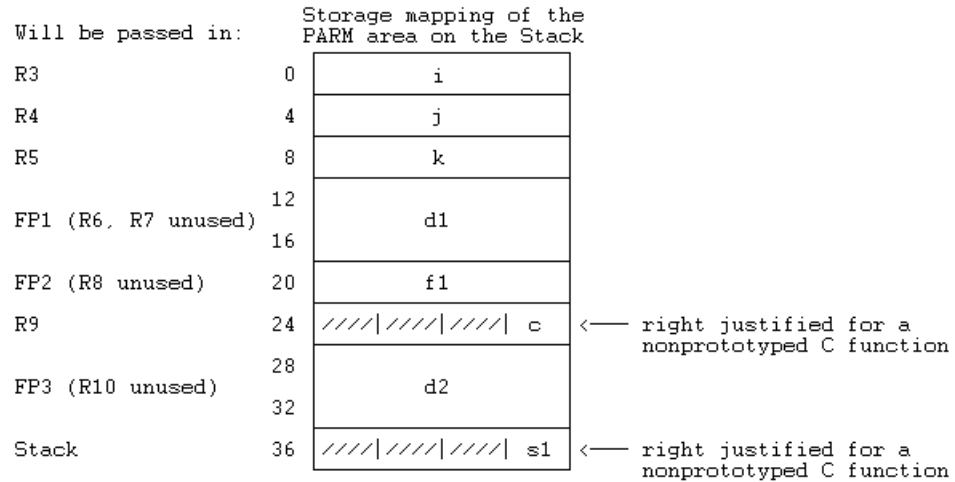


```

char c;
...
void f(int, int, int, double, float, char, double, short);
f( i, j, k, d1, f1, c, d2, s1 );

```

The function call results in the following storage mapping:



Notes:

1. A parameter is guaranteed to be mapped only if its address is taken.
2. Data with less than fullword alignment is copied into high-order bytes. Because the function in the example is prototyped, the mapping of parameters c and s1 is right-justified.
3. The parameter list is a conceptually contiguous piece of storage containing a list of words. For efficiency, the first 8 words of the list are not actually stored in the space reserved for them, but passed in GPR3-GPR10. Furthermore, the first 13 floating point value parameter values are not passed in GPRs, but are passed in FPR1-FPR13. In all cases, parameters beyond the first 8 words of the list are also stored in the space reserved for them.
4. If the called procedure intends to treat the parameter list as a contiguous piece of storage (for example, if the address of a parameter is taken in C), the parameter registers are stored in the space reserved for them in the stack.
5. A register image is stored on the stack.
6. The argument area (P₁ ... P_n) must be large enough to hold the largest parameter list.

Related Concepts

“Using VisualAge C++ with Other Programming Languages” on page 15

Related Tasks

- “Interlanguage Calling Conventions” on page 47
- “Corresponding Data Types” on page 47
- “Use the Subroutine Linkage Conventions in Interlanguage Calls” on page 49
- “Sample Program: C Calling Fortran” on page 57
- “Interlanguage Calls - Parameter Passing” on page 50
- “Interlanguage Calls - Call by Reference Parameters” on page 51
- “Interlanguage Calls - Call by Value Parameters” on page 52
- “Interlanguage Calls - Pointers to Functions” on page 54
- “Interlanguage Calls - Function Return Values” on page 54

- “Interlanguage Calls - Stack Floor” on page 55
- “Interlanguage Calls - Stack Overflow” on page 55
- “Interlanguage Calls - Traceback Table” on page 56
- “Interlanguage Calls - Type Encoding and Checking” on page 56

Interlanguage Calls - Pointers to Functions

A function pointer is a data type whose values range over function addresses. Variables of this type appear in several programming languages such as C and Fortran. In Fortran, a dummy argument that appears in an **EXTERNAL** statement is a function pointer. Function pointers are supported in contexts such as the target of a call statement or an actual argument of such a statement.

A function pointer is a fullword quantity that is the address of a function descriptor. The function descriptor is a 3-word object. The first word contains the address of the entry point of the procedure, the second has the address of the TOC of the module in which the procedure is bound, and the third is the environment pointer for languages such as Pascal. There is only one function descriptor per entry point. It is bound into the same module as the function it identifies, if the function is external. The descriptor has an external name, which is the same as the function name, but without a leading . (dot). This descriptor name is used in all import and export operations.

Related Concepts

“Using VisualAge C++ with Other Programming Languages” on page 15

Related Tasks

- “Interlanguage Calling Conventions” on page 47
- “Corresponding Data Types” on page 47
- “Use the Subroutine Linkage Conventions in Interlanguage Calls” on page 49
- “Sample Program: C Calling Fortran” on page 57
- “Interlanguage Calls - Parameter Passing” on page 50
- “Interlanguage Calls - Call by Reference Parameters” on page 51
- “Interlanguage Calls - Call by Value Parameters” on page 52
- “Interlanguage Calls - Rules for Passing Parameters by Value” on page 52
- “Interlanguage Calls - Function Return Values”
- “Interlanguage Calls - Stack Floor” on page 55
- “Interlanguage Calls - Stack Overflow” on page 55
- “Interlanguage Calls - Traceback Table” on page 56
- “Interlanguage Calls - Type Encoding and Checking” on page 56

Interlanguage Calls - Function Return Values

Functions pass their return values according to type:

- Pointers, enumerated types, and integral values (**int**, **short**, **long**, **char**, and unsigned types) of any length are returned, right-justified, in R3; **long long** values are returned in R3 and R4. (R3 in 64-bit mode)
- **floats** and **doubles** are returned in FP1; 128-bit **long doubles** are returned in FP1 and FP2.
- Calling functions supply a pointer to a memory location where the called function stores the returned value.
- **long doubles** are returned in R1 and R2.

Related Concepts

“Using VisualAge C++ with Other Programming Languages” on page 15

Related Tasks

- “Interlanguage Calling Conventions” on page 47
- “Corresponding Data Types” on page 47
- “Use the Subroutine Linkage Conventions in Interlanguage Calls” on page 49
- “Sample Program: C Calling Fortran” on page 57
- “Interlanguage Calls - Parameter Passing” on page 50
- “Interlanguage Calls - Call by Reference Parameters” on page 51
- “Interlanguage Calls - Call by Value Parameters” on page 52
- “Interlanguage Calls - Rules for Passing Parameters by Value” on page 52
- “Interlanguage Calls - Pointers to Functions” on page 54
- “Interlanguage Calls - Stack Floor”
- “Interlanguage Calls - Stack Overflow”
- “Interlanguage Calls - Traceback Table” on page 56
- “Interlanguage Calls - Type Encoding and Checking” on page 56

Interlanguage Calls - Stack Floor

The *stack floor* is a system-defined address below which the stack cannot grow.

Other system invariants related to the stack must be maintained by all compilers and assemblers:

- No data is saved or accessed from an address lower than the stack floor.
- The stack pointer is always valid. When the stack frame size is more than 32767 bytes, take care to ensure that its value is changed in a single instruction, so that there is no timing window in which a signal handler would either overlay the stack data or erroneously appear to overflow the stack segment.

Related Concepts

- “Using VisualAge C++ with Other Programming Languages” on page 15

Related Tasks

- “Interlanguage Calling Conventions” on page 47
- “Corresponding Data Types” on page 47
- “Use the Subroutine Linkage Conventions in Interlanguage Calls” on page 49
- “Sample Program: C Calling Fortran” on page 57
- “Interlanguage Calls - Parameter Passing” on page 50
- “Interlanguage Calls - Call by Reference Parameters” on page 51
- “Interlanguage Calls - Call by Value Parameters” on page 52
- “Interlanguage Calls - Rules for Passing Parameters by Value” on page 52
- “Interlanguage Calls - Pointers to Functions” on page 54
- “Interlanguage Calls - Function Return Values” on page 54
- “Interlanguage Calls - Stack Overflow”
- “Interlanguage Calls - Traceback Table” on page 56
- “Interlanguage Calls - Type Encoding and Checking” on page 56

Interlanguage Calls - Stack Overflow

The RISC System/6000 linkage convention requires no explicit inline check for overflow. The operating system uses a storage-protect mechanism to detect stores past the end of the stack segment.

Related Concepts

- “Using VisualAge C++ with Other Programming Languages” on page 15

Related Tasks

- “Interlanguage Calling Conventions” on page 47
- “Corresponding Data Types” on page 47

“Use the Subroutine Linkage Conventions in Interlanguage Calls” on page 49
“Sample Program: C Calling Fortran” on page 57
“Interlanguage Calls - Parameter Passing” on page 50
“Interlanguage Calls - Call by Reference Parameters” on page 51
“Interlanguage Calls - Call by Value Parameters” on page 52
“Interlanguage Calls - Rules for Passing Parameters by Value” on page 52
“Interlanguage Calls - Pointers to Functions” on page 54
“Interlanguage Calls - Function Return Values” on page 54
“Interlanguage Calls - Stack Floor” on page 55
“Interlanguage Calls - Traceback Table”
“Interlanguage Calls - Type Encoding and Checking”

Interlanguage Calls - Traceback Table

The compiler supports the traceback mechanism, which is required by the AIX Operating System symbolic debugger to unravel the call or return stack. Each function has a traceback table in the text segment at the end of its code. This table contains information about the function, including the type of function as well as stack frame and register information.

Related Concepts

“Using VisualAge C++ with Other Programming Languages” on page 15

Related Tasks

“Interlanguage Calling Conventions” on page 47
“Corresponding Data Types” on page 47
“Use the Subroutine Linkage Conventions in Interlanguage Calls” on page 49
“Sample Program: C Calling Fortran” on page 57
“Interlanguage Calls - Parameter Passing” on page 50
“Interlanguage Calls - Call by Reference Parameters” on page 51
“Interlanguage Calls - Call by Value Parameters” on page 52
“Interlanguage Calls - Rules for Passing Parameters by Value” on page 52
“Interlanguage Calls - Pointers to Functions” on page 54
“Interlanguage Calls - Function Return Values” on page 54
“Interlanguage Calls - Stack Floor” on page 55
“Interlanguage Calls - Stack Overflow” on page 55
“Interlanguage Calls - Type Encoding and Checking”

Interlanguage Calls - Type Encoding and Checking

Detecting errors before a program is run is a key objective of IBM VisualAge C++. Runtime errors are hard to find, and many are caused by mismatching subroutine interfaces or conflicting data definitions.

VisualAge C++ uses a scheme for early detection that encodes information about all external symbols (data and programs). If the `-qextchk` option has been specified, this information about external symbols is checked at bind or load time for consistency.

The *AIX 5L for POWER-based Systems: Assembler Language Reference* book describes the following details of the Subroutine Linkage Convention:

- Register usage (general-purpose, floating-point, and special-purpose registers)
- Stack
- The calling routine’s responsibilities
- The called routine’s responsibilities

Related Concepts

"Using VisualAge C++ with Other Programming Languages" on page 15

Related Tasks

"Interlanguage Calling Conventions" on page 47

"Corresponding Data Types" on page 47

"Use the Subroutine Linkage Conventions in Interlanguage Calls" on page 49

"Sample Program: C Calling Fortran"

"Interlanguage Calls - Parameter Passing" on page 50

"Interlanguage Calls - Call by Reference Parameters" on page 51

"Interlanguage Calls - Call by Value Parameters" on page 52

"Interlanguage Calls - Rules for Passing Parameters by Value" on page 52

"Interlanguage Calls - Pointers to Functions" on page 54

"Interlanguage Calls - Function Return Values" on page 54

"Interlanguage Calls - Stack Floor" on page 55

"Interlanguage Calls - Stack Overflow" on page 55

"Interlanguage Calls - Traceback Table" on page 56

Also, on the Web see:

AIX 5L for POWER-based Systems: Assembler Language Reference
Files Reference

Sample Program: C Calling Fortran

A C program can call a Fortran function or subroutine.

The following example illustrates how program units written in different languages can be combined to create a single program. It also demonstrates parameter passing between C and Fortran subroutines with different data types as arguments.

```
#include <iostream.h>
extern double add(int *, double [],
int *, double []);

double ar1[4]={1.0, 2.0, 3.0, 4.0};
double ar2[4]={5.0, 6.0, 7.0, 8.0};

main()
{
int x, y;
double z;

x = 3;

z = add(&x, ar1, y, ar2); /* Call Fortran add routine */
/* Note: Fortran indexes arrays 1..n*/
/* C indexes arrays 0..(n-1) */

printf("The sum of %1.0f and %1.0f is %2.0f \n",
ar1[x-1], ar2[y-1], z);
}
```

The Fortran subroutine is:

```
C Fortran function add.f - for C interlanguage call example
C Compile separately, then link to C program
```

```
REAL FUNCTION ADD*8 (A, B, C, D)
REAL*8 B,D
INTEGER*4 A,C
```

```
DIMENSION B(4), D(4)
ADD = B(A) + D(C)
RETURN
END
```

Related Concepts

“Using VisualAge C++ with Other Programming Languages” on page 15

Related Tasks

“Interlanguage Calling Conventions” on page 47

“Corresponding Data Types” on page 47

“Use the Subroutine Linkage Conventions in Interlanguage Calls” on page 49

Part 3. Reference

Compiler Options

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

- “Compiler Command Line Options”
- “General Purpose Pragmas” on page 297
- “Pragmas to Control Parallel Processing” on page 344


Compiler Command Line Options





This section lists and describes VisualAge C++ command line options.



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



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



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







Option Name	Type	Default	Description
+ (plus sign)	<i>-flag</i>	-	 Compiles any file, <i>filename.nnn</i> , as a C++ language file, where <i>nnn</i> is any suffix other than .o , .a , or .s .
# (pound sign)	<i>-flag</i>	-	Traces the compilation without doing anything.
32, 64	<i>-qopt</i>	32	Selects 32- or 64-bit compiler mode.
aggrcopy	<i>-qopt</i>	See aggrcopy .	Enables destructive copy operations for structures and unions.
alias	<i>-qopt</i>	See alias .	Specifies which type-based aliasing is to be used during optimization.
align	<i>-qopt</i>	align=full	Specifies what aggregate alignment rules the compiler uses for file compilation.
alloca	<i>-qopt</i>	-	Substitutes inline code for calls to function alloca as if #pragma alloca directives are in the source code.
ansialias	<i>-qopt</i>	See ansialias .	Specifies whether type-based aliasing is to be used during optimization.
arch	<i>-qopt</i>	arch=com	Specifies the architecture on which the executable program will be run.

Option Name	Type	Default	Description
assert	<i>-qopt</i>	noassert	 Requests the compiler to apply aliasing assertions to your compilation unit.
attr	<i>-qopt</i>	noattr	Produces a compiler listing that includes an attribute listing for all identifiers.
B	<i>-flag</i>	-	Determines substitute path names for the compiler, assembler, linkage editor, and preprocessor.
b	<i>-flag</i>	bdynamic	Instructs the linker to process subsequent shared objects as either dynamic, shared or static.
bitfields	<i>-qopt</i>	unsigned	Specifies if bitfields are signed.
bmaxdata	<i>-flag</i>	0	Sets the size of the heap in bytes.
brtl	<i>-flag</i>	-	Enables runtime linking.
C	<i>-flag</i>	-	Preserves comments in preprocessed output.
c	<i>-flag</i>	-	Instructs the compiler to pass source files to the compiler only.
cache	<i>-qopt</i>	-	Specify a cache configuration for a specific execution machine.
chars	<i>-qopt</i>	chars=unsigned	Instructs the compiler to treat all variables of type char as either signed or unsigned .
check	<i>-qopt</i>	nocheck	Generates code which performs certain types of run-time checking.
cinc	<i>-qopt</i>	nocinc	 Include files from specified directories have the tokens extern "C" { inserted before the file, and } appended after the file.
compact	<i>-qopt</i>	nocompact	When used with optimization, reduces code size where possible, at the expense of execution speed.
cpluscmt	<i>-qopt</i>	See cpluscmt .	 Use this option if you want C++ comments to be recognized in C source files.
D	<i>-flag</i>	-	Defines the identifier <i>name</i> as in a #define preprocessor directive.
dataimported	<i>-qopt</i>	-	Mark data as imported.
datalocal	<i>-qopt</i>	-	Marks data as local.
dbxextra	<i>-qopt</i>	nodbxextra	 Specifies that all typedef declarations, struct , union , and enum type definitions are included for debugger processing.
digraph	<i>-qopt</i>	See digraph .	Allows use of digraph character sequences in your program.
dollar	<i>-qopt</i>	nodollar	Allows the \$ symbol to be used in the names of identifiers.

Option Name	Type	Default	Description
dpcl	-qopt	nodpcl	Generates block scopes to support the IBM Dynamic Probe Class Library.
E	<i>-flag</i>	-	Runs the source files named in the compiler invocation through the preprocessor.
e	<i>-flag</i>	-	Specifies the entry name for the shared object. Equivalent to using ld -e name . See your system documentation for additional information about ld options.
eh	-qopt	eh	 Controls exception handling.
enum	-qopt	enum=int	Specifies the amount of storage occupied by the enumerations.
expfile	-qopt	-	Saves all exported symbols in a file.
extchk	-qopt	noextchk	Generates bind-time type checking information and checks for compile-time consistency.
F	<i>-flag</i>	-	Names an alternative configuration file for x1C .
f	<i>-flag</i>	-	Names a file to store a list of object files.
fdpr	-qopt	nofdpr	Collect program information for use with the AIX fdpr performance-tuning utility.
flag	-qopt	flag=i:i	Specifies the minimum severity level of diagnostic messages to be reported.
float	-qopt	See float .	Specifies various floating point options to speed up or improve the accuracy of floating point operations.
flttrap	-qopt	noflttrap	Generates extra instructions to detect and trap floating point exceptions.
fold	-qopt	fold	Specifies that constant floating point expressions are to be evaluated at compile time.
fullpath	-qopt	nofullpath	Specifies what path information is stored for files when you use -g and the distributed graphical debugger.
funcsect	-qopt	nofuncsect	Place instructions for each function in a separate object file, control section or csect.
G	<i>-flag</i>	-	Linkage editor (ld command) option only. Used to generate a dynamic library file.
g	<i>-flag</i>	-	Generates debugging information used by a debugger such as the Distributed Debugger.
genproto	-qopt	nogenproto	 Produces ANSI prototypes from K&R function definitions.
halt	-qopt	halt=s	Instructs the compiler to stop after the compilation phase when it encounters errors of specified <i>severity</i> or greater.

Option Name	Type	Default	Description
haltonmsg	<code>-qopt</code>	-	 Instructs the compiler to stop after the compilation phase when it encounters a specific error message.
heapdebug	<code>-qopt</code>	noheapdebug	Enables debug versions of memory management functions.
hot	<code>-qopt</code>	nohot	Instructs the compiler to perform high-order transformations on loops and array language during optimization, and to pad array dimensions and data objects to avoid cache misses.
hsflt	<code>-qopt</code>	nohsflt	Speeds up calculations by removing range checking on single-precision float results and on conversions from floating point to integer.
hssngl	<code>-qopt</code>	nohssngl	Specifies that single-precision expressions are rounded only when the results are stored into float memory locations.
I	<code>-flag</code>	-	Specifies an additional search path if the file name in the #include directive is not specified using its absolute path name.
idirfirst	<code>-qopt</code>	noidirfirst	Specifies the search order for files included with the #include "file_name" directive.
ignerrno	<code>-qopt</code>	noignerrno	Allows the compiler to perform optimizations that assume errno is not modified by system calls.
ignprag	<code>-qopt</code>	-	Instructs the compiler to ignore certain pragma statements.
info	<code>-qopt</code>	noinfo	Produces informational messages.
initauto	<code>-qopt</code>	noinitauto	Initializes automatic storage to a specified two-digit hexadecimal byte value.
inlglue	<code>-qopt</code>	noinlglue	Generates fast external linkage by inlining the pointer glue code necessary to make a call to an external function or a call through a function pointer.
inline	<code>-qopt</code>	See inline .	Attempts to inline functions instead of generating calls to a function.
ipa	<code>-qopt</code>	ipa=object (<i>compile-time</i>) noipa (<i>link-time</i>)	Turns on or customizes a class of optimizations known as interprocedural analysis (IPA).
isolated_call	<code>-qopt</code>	-	Specifies functions in the source file that have no side effects.
keepinlines	<code>-qopt</code>	nokeepinlines	 Instructs the compiler to keep or discard definitions for unreferenced extern inline functions.

Option Name	Type	Default	Description
keyword	<i>-qopt</i>	See keyword .	Controls whether a specified string is treated as a keyword or an identifier.
L	<i>-flag</i>	See L .	Searches the specified directory for library files specified by the -l option.
l	<i>-flag</i>	See l .	Searches a specified library for linking.
langlvl	<i>-qopt</i>	See langlvl .	Selects the C or C++ language level for compilation.
largepage	<i>-qopt</i>	nolargepage .	Instructs the compiler to exploit large page heaps available on Power 4 systems running AIX v5.1D or later.
ldbl128, longdouble	<i>-qopt</i>	noldbl128	Increases the size of long double type from 64 bits to 128 bits.
libansi	<i>-qopt</i>	nolibansi	Assumes that all functions with the name of an ANSI C library function are in fact the system functions.
linedebug	<i>-qopt</i>	nolinedebug	Generates abbreviated line number and source file name information for the debugger.
list	<i>-qopt</i>	nolist	Produces a compiler listing that includes an object listing.
listopt	<i>-qopt</i>	nolistopt	Produces a compiler listing that displays all options in effect.
longlit	<i>-qopt</i>	nolonglit	Makes unsuffixed literals the long type for 64-bit mode.
longlong	<i>-qopt</i>	See longlong .	Allows long long types in your program.
M	<i>-flag</i>	-	Creates an output file that contains targets suitable for inclusion in a description file for the AIX make command.
ma	<i>-flag</i>	-	 Substitutes inline code for calls to function alloca as if #pragma alloca directives are in the source code.
macpstr	<i>-qopt</i>	nomacpstr	 Converts Pascal string literals into null-terminated strings where the first byte contains the length of the string.
maf	<i>-qopt</i>	maf	Specifies whether the floating-point multiply-add instructions are to be generated.
makedep	<i>-qopt</i>	-	Creates an output file that contains targets suitable for inclusion in a description file for the AIX make command.
maxerr	<i>-qopt</i>	nomaxerr	Instructs the compiler to halt compilation when a specified number of errors of specified or greater severity is reached.

Option Name	Type	Default	Description
maxmem	<i>-qopt</i>	maxmem=8192	Limits the amount of memory used for local tables of specific, memory-intensive optimizations.
mbscs, dbscs	<i>-qopt</i>	nombcs	Use the -qmbcs option if your program contains multibyte characters.
mkshrojb	<i>-qopt</i>	-	Creates a shared object from generated object files.
namemangling	<i>-qopt</i>	namemangling=ansi	 Selects the name mangling scheme for external symbol names generated from C++ source code.
O, optimize	<i>-qopt, -flag</i>	nooptimize	Optimizes code at a choice of levels during compilation.
o	<i>-flag</i>	-	Specifies a name or directory for the output executable file(s) created either by the compiler or the linkage editor.
objmodel	<i>-qopt</i>	objmodel=compat	 Sets the type of object model.
oldpassbyvalue	<i>-qopt</i>	nooldpassbyvalue	 Specifies how classes containing const or reference members are passed in function arguments.
P	<i>-flag</i>	-	Preprocesses the C or C++ source files named in the compiler invocation and creates an output preprocessed source file for each input source file.
p	<i>-flag</i>	-	Sets up the object files produced by the compiler for profiling.
pascal	<i>-qopt</i>	nopascal	 Ignores the word pascal in type specifiers and function declarations.
path	<i>-qopt</i>	-	Constructs alternate program and path names.
pdf1, pdf2	<i>-qopt</i>	nopdf1, nopdf2	Tunes optimizations through Profile-Directed Feedback.
pg	<i>-flag</i>	-	Sets up the object files for profiling, but provides more information than is provided by the -p option.
phsinfo	<i>-qopt</i>	nophsinfo	Reports the time taken in each compilation phase.
print	<i>-qopt</i>	-	Suppresses listings.
priority	<i>-qopt</i>	-	 Specifies the priority level for the initialization of static constructors
proclcal, procimported, procunknown	<i>-qopt</i>	See proclcal .	Mark functions as local, imported, or unknown.
proto	<i>-qopt</i>	noproto	 Assumes all functions are prototyped.
Q	<i>-flag</i>	See Q .	Attempts to inline functions instead of generating calls to a function.
r	<i>-flag</i>	-	Produces a relocatable object.

Option Name	Type	Default	Description
report	<i>-qopt</i>	noreport	Instructs the compiler to produce transformation reports that show how program loops are parallelized and optimized.
rndflt	<i>-qopt</i>	norndflt	Controls the compile-time rounding mode of constant floating point expressions.
rndsngl	<i>-qopt</i>	norndsngl	Specifies that the result of each single-precision float operation is to be rounded to single precision.
ro	<i>-qopt</i>	See ro .	Specifies the storage type for string literals.
roconst	<i>-qopt</i>	See roconst .	Specifies the storage location for constant values.
rrm	<i>-qopt</i>	norm	Prevents floating-point optimizations that are incompatible with run-time rounding to plus and minus infinity modes.
rtti	<i>-qopt</i>	nortti	► C++ Generates run-time type identification (RTTI) information for the typeid operator and the dynamic_cast operator.
S	<i>-flag</i>	-	Generates an assembly language file (.s) for each source file.
s	<i>-flag</i>	-	Strips symbol table.
showinc	<i>-qopt</i>	noshowinc	If used with -qsource , all the include files are included in the source listing.
smallstack	<i>-qopt</i>	nosmallstack	Instructs the compiler to reduce the size of the stack frame.
smp	<i>-qopt</i>	nosmp	Enables parallelization of IBM SMP-compliant program code.
source	<i>-qopt</i>	nosource	Produces a compiler listing and includes source code.
spill	<i>-qopt</i>	spill=512	Specifies the size of the register allocation spill area.
spnans	<i>-qopt</i>	nospnans	Generates extra instructions to detect signalling NaN on conversion from single precision to double precision.
srcmsg	<i>-qopt</i>	nosrcmsg	► C Adds the corresponding source code lines to the diagnostic messages in the stderr file.
staticinline	<i>-qopt</i>	nostaticinline	► C++ Controls whether inline functions are treated as static or extern.
statsym	<i>-qopt</i>	nostatsym	Adds user-defined, non-external names that have a persistent storage class to the name list.
stdinc	<i>-qopt</i>	stdinc	Specifies which files are included with #include <file_name> and #include "file_name" directives.

Option Name	Type	Default	Description
strict	<i>-qopt</i>	See strict .	Turns off aggressive optimizations of the -O3 option that have the potential to alter the semantics of your program.
strict_induction	<i>-qopt</i>	See strict_induction .	Disables loop induction variable optimizations that have the potential to alter the semantics of your program.
suppress	<i>-qopt</i>	nosuppress	Specifies compiler message numbers to be suppressed.
symtab	<i>-qopt</i>	-	Set symbol tables for unreferenced variables or xcoff objects.
syntaxonly	<i>-qopt</i>	-	C Causes the compiler to perform syntax checking without generating an object file.
t	<i>-flag</i>	See t .	Adds the prefix specified by the -B option to designated programs. -tE replaces the CreateExportList script.
tabsize	<i>-qopt</i>	tabsize=8	Changes the length of tabs as perceived by the compiler.
thtable	<i>-qopt</i>	See thtable=full .	Sets traceback table characteristics.
tempinc	<i>-qopt</i>	See tempinc .	C++ Generates separate include files for template functions and class declarations, and places these files in a directory which can be optionally specified.
templaterecompile	<i>-qopt</i>	See templaterecompile .	C++ Helps manage dependencies between compilation units that have been compiled using the -qtemplateregistry compiler option.
templateregistry	<i>-qopt</i>	See templateregistry .	C++ Maintains records of all templates as they are encountered in the source and ensures that only one instantiation of each template is made.
tempmax	<i>-qopt</i>	tempmax=1	C++ Specifies the maximum number of template include files to be generated by the tempinc option for each header file.
threaded	<i>-qopt</i>	See threaded .	Indicates that the program will run in a multi-threaded environment.
tmplparse	<i>-qopt</i>	tmplparse=no	C++ Controls whether parsing and semantic checking are applied to template definition implementations.
tocdata	<i>-qopt</i>	notocdata .	Marks data as local.
tocmerge	<i>-qopt</i>	notocmerge .	Enables TOC merging to reduce TOC pointer loads and improves the scheduling of external loads.
tune	<i>-qopt</i>	See tune .	Specifies the architecture for which the executable program is optimized.
twolink	<i>-qopt</i>	See notwolink .	C++ Minimizes the number of static constructors included from libraries.

Option Name	Type	Default	Description
U	<i>-flag</i>	-	Undefines a specified identifier defined by the compiler or by the -D option.
unique	-qopt	nounique	C++ Generates unique names for static constructor/destructor file compilation units.
unroll	-qopt	unroll=auto	Unrolls inner loops in the program.
unwind	-qopt	unwind	Informs the compiler that the application does not rely on any program stack unwinding mechanism.
upconv	-qopt	noupconv	C Preserves the unsigned specification when performing integral promotions.
V	<i>-flag</i>	-	Instructs the compiler to report information on the progress of the compilation in a command-like format.
v	<i>-flag</i>	-	Instructs the compiler to report information on the progress of the compilation.
vftable	-qopt	See vftable .	C++ Controls the generation of virtual function tables.
W	<i>-flag</i>	-	Passes the listed words to a designated compiler program.
w	<i>-flag</i>	-	Requests that warning messages be suppressed.
warn64	-qopt	nowarn64	Enables warning of possible long to integer data truncations.
xcall	-qopt	noxcall	Generates code to static routines within a compilation unit as if they were external calls.
xref	-qopt	noxref	Produces a compiler listing that includes a cross-reference listing of all identifiers.
y	<i>-flag</i>	-	Specifies the compile-time rounding mode of constant floating-point expressions. See also rndflt .
Z	<i>-flag</i>	-	Specifies a search path for library names.

Related Concepts

“Compiler Options” on page 5

Related Tasks

“Specify Compiler Options on the Command Line” on page 25

“Specify Compiler Options in Your Program Source Files” on page 27

“Specify Compiler Options in a Configuration File” on page 27

“Specify Compiler Options for Architecture-Specific, 32- or 64-bit Compilation” on page 29

“Resolving Conflicting Compiler Options” on page 31

Related References

"General Purpose Pragmas" on page 297

"Pragmas to Control Parallel Processing" on page 344

"Acceptable Compiler Mode and Processor Architecture Combinations" on page 373

+ (plus sign)

C++

Purpose

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

Syntax

▶▶ — -+ —————▶▶

Notes

If you do not use the **-+** option, files must have a suffix of **.C** (uppercase C), **.cc**, **.cpp**, or **.cxx** to be compiled as a C++ file. If you compile files with suffix **.c** (lowercase c) without specifying **-+**, the files are compiled as a C language file.

Example

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

```
x1C -+ myprogram.cplspls
```

Related References

“Compiler Command Line Options” on page 61

(pound sign)



Purpose

Traces the compilation without invoking anything. This option previews the compilation steps specified on the command line. When the `x1C` command is issued with this option, it names the programs within the preprocessor, compiler, and linkage editor that would be invoked, and the options that would be specified to each program. The preprocessor, compiler, and linkage editor are not invoked.

Syntax

▶▶ — `-#` —————▶▶

Notes

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

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

Example

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

```
x1C myprogram.c -#
```

Related References

“Compiler Command Line Options” on page 61

“v” on page 288

32, 64

C C++

Purpose

Selects either 32- or 64-bit compiler mode.

Syntax

► — -q — 32 —————►
└──┬──┘
64

Notes

The **-q32** and **-q64** options override the compiler mode set by the value of the `OBJECT_MODE` environment variable, if it exists. If the **-q32** and **-q64** options are not specified, and the `OBJECT_MODE` environment variable is not set, the compiler defaults to 32-bit output mode.

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

Use **-q32** and **-q64** options, along with the **-qarch** and **-qtune** compiler options, to optimize the output of the compiler to the architecture on which that output will be used. Refer to the *Acceptable Compiler Mode and Processor Architecture Combinations* table for valid combinations of the **-q32**, **-q64**, **-qarch**, and **-qtune** compiler options. In 64-bit mode, **-qarch=com** is treated the same as **-qarch=ppc**.

Using **-qarch=ppc** or any ppc family architecture with **-qfloat=hssngl** or **-qfloat=hsflt** may produce incorrect results on rs64b or future systems.

Example

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

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

Important Notes!

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

Related References

“Compiler Command Line Options” on page 61

“arch” on page 83

“float” on page 131

“tune” on page 277

“Acceptable Compiler Mode and Processor Architecture Combinations” on page 373

aggrcopy

> C > C++

Purpose

Enables destructive copy operations for structures and unions.

Syntax

►► -q-aggrcopy=nooverlap | overlap ►►

Default Setting

The default setting of this option is **-qaggrcopy=nooverlap** when compiling to the ANSI, SAA, SAAL2, extc89, stdc99, and extc99 language levels.

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

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

Notes

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

Example

```
x1C myprogram.c -qaggrcopy=nooverlap
```

Related References

“Compiler Command Line Options” on page 61

“langlvl” on page 175

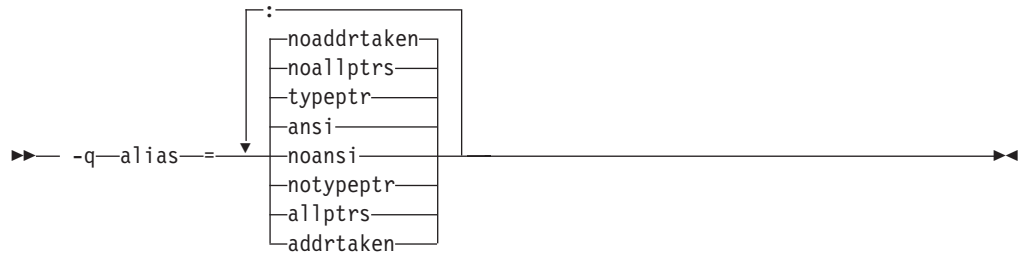
alias

C C++

Purpose

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

Syntax



where available aliasing options are:

- | | |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [NO]TYPEptr | Pointers to different types are never aliased. In other words, in the compilation unit no two pointers of different types will point to the same storage location. |
| [NO]ALLPtrs | Pointers are never aliased (this also implies <code>-qalias=typeptr</code>). Therefore, in the compilation unit, no two pointers will point to the same storage location. |
| [NO]ADDRtaken | Variables are disjoint from pointers unless their address is taken. Any class of variable for which an address has <i>not</i> been recorded in the compilation unit will be considered disjoint from indirect access through pointers. |
| [NO]ANSI | Type-based aliasing is used during optimization, which restricts the lvalues that can be safely used to access a data object. The optimizer assumes that pointers can <i>only</i> point to an object of the same type. This (<code>ansi</code>) is the default for the <code>xlC</code> , <code>xlC</code> , and <code>c89</code> compilers. This option has no effect unless you also specify the <code>-O</code> option. |

If you select `noansi`, the optimizer makes worst case aliasing assumptions. It assumes that a pointer of a given type can point to an external object or any object whose address is already taken, regardless of type. This is the default for the `cc` compiler.

Notes

The following are not subject to type-based aliasing:

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

Example

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

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

Related References

"Compiler Command Line Options" on page 61

"ansialias" on page 82

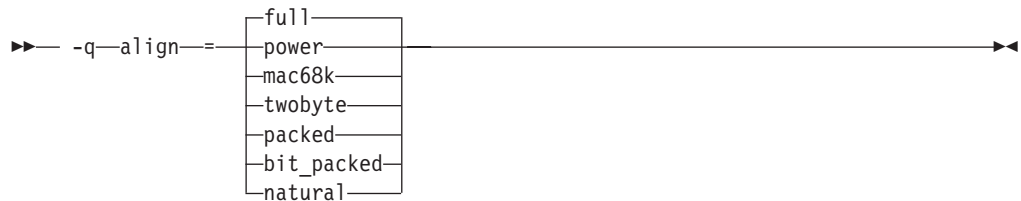
align

C C++

Purpose

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

Syntax



where available alignment options are:

<code>power</code>	The compiler uses the RISC System/6000 alignment rules.
<code>full</code>	The compiler uses the RISC System/6000. alignment rules. The <i>power</i> option is the same as <i>full</i> .
<code>mac68k</code>	The compiler uses the Macintosh** alignment rules.
<code>twobyte</code>	The compiler uses the Macintosh alignment rules. The <i>mac68k</i> option is the same as <i>twobyte</i> .
<code>packed</code>	The compiler uses the packed alignment rules.
<code>bit_packed</code>	The compiler uses the bit_packed alignment rules. Alignment rules for bit_packed are the same as that for packed alignment except that bitfield data is packed on a bit-wise basis without respect to byte boundaries.
<code>natural</code>	The compiler maps structure members to their natural boundaries. This has the same effect as the <i>power</i> suboption, except that it also applies alignment rules to doubles and long doubles that are not the first member of a structure or union.

See also “`#pragma align`” on page 299 and “`#pragma options`” on page 325.

Notes

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

Within your source file, you can use `#pragma options align=reset` to revert to a previous alignment rule. The compiler stacks alignment directives, so you can go back to using the previous alignment directive, without knowing what it is, by specifying the `#pragma align=reset` directive. For example, you can use this option if you have a class declaration within an include file and you do not want the alignment rule specified for the class to apply to the file in which the class is included.

You can code `#pragma options align=reset` in a source file to change the alignment option to what it was before the last alignment option was specified. If no previous alignment rule appears in the file, the alignment rule specified in the invocation command is used.

Examples

Example 1 - Imbedded #pragmas

Using the compiler invocation:

```
x1C -qalign=mac68k file.c /* <-- default alignment rule for file is */
                          /*   Macintosh                          */
```

Where file.c has:

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

Example 2 - Affecting Only Aggregate Definition

Using the compiler invocation:

```
x1C file2.c /* <-- default alignment rule for file is                */
           /*   RISC System/6000 since no alignment rule specified */
```

Where file2.c has:

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

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

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

Using the `__align` specifier

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

`__align` Specifier:

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

Syntax:

```
declarator __align (int_const) identifier;

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

where:

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

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

The `__align` specifier cannot be used on individual elements within an aggregate definition, but it can be used on an aggregate definition nested within another aggregate definition.

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

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

Not all alignments may be representable in an object file.

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

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

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

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

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

```
__align(128) struct S {int i;}; /* sizeof(struct S) == 128          */
struct S sarray[10];           /* sarray is aligned on 128-byte boundary
                             with sizeof(sarray) == 1280      */
struct S __align(64) svar;     /* error - alignment of variable is
                             smaller than alignment of type      */
struct S2 {struct S s1; int a;} s2; /* s2 is aligned on 128-byte boundary
                             with sizeof(s2) == 256 bytes       */
```

Applying `__align` to an array:

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

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

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

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

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

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

int __align(1) s4;         /* error */

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

Related References

“Compiler Command Line Options” on page 61

“#pragma align” on page 299

alloca

► C ► C++

Purpose

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

Syntax

►► -q—alloca◄◄

Notes

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

C++ programs must specify `#include <malloc.h>` to include the `alloca` function declaration.

Example

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

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

Related References

“Compiler Command Line Options” on page 61

“ma” on page 199

“#pragma alloca” on page 300

ansialias

C C++

Purpose

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

Syntax

► -q ansialias
noansialias ◄

See also “#pragma options” on page 325.

Notes

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

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

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

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

If you select **noansialias**, the optimizer makes worst-case aliasing assumptions. It assumes that a pointer of a given type can point to an external object or any object whose address is already taken, regardless of type.

The following are not subject to type-based aliasing:

- Signed and unsigned types; for example, a pointer to a **signed int** can point to an **unsigned int**.
- Character pointer types can point to any type.
- Types qualified as **volatile** or **const**; for example, a pointer to a **const int** can point to an **int**.

Example

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

```
xlC myprogram.c -O -qnoansialias
```

Related References

“Compiler Command Line Options” on page 61

“alias” on page 75

“#pragma options” on page 325

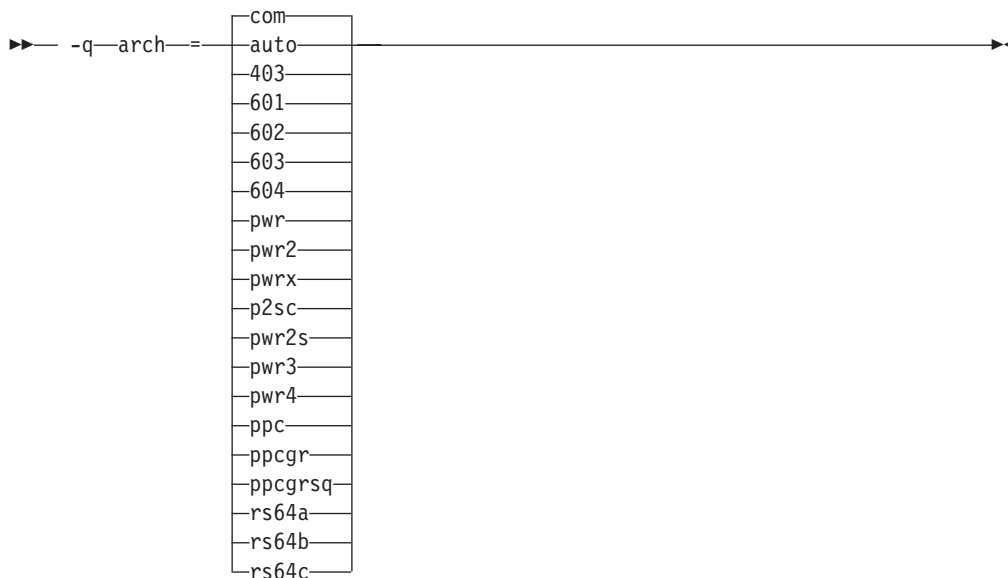
arch

C C++

Purpose

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

Syntax



where available architecture options are:

- | | |
|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| com | <ul style="list-style-type: none">• In 32-bit execution mode, produces object code containing instructions that will run on any of the POWER, POWER2*, and PowerPC* hardware platforms (that is, the instructions generated are <i>common</i> to all platforms. Using -qarch=com is referred to as compiling in <i>common mode</i>.• In 64-bit mode, produces object code that will run on all the 64-bit PowerPC hardware platforms but not 32-bit-only platforms.• Defines the <code>_ARCH_COM</code> macro and produces portable programs.• This is the default option unless the -O4 compiler option was specified. |
| auto | <ul style="list-style-type: none">• Produces object code containing instructions that will run on the hardware platform on which it is compiled. |
| PPC | <ul style="list-style-type: none">• In 32-bit mode, produces object code containing instructions that will run on any of the 32-bit PowerPC hardware platforms. This suboption will cause the compiler to produce single-precision instructions to be used with single-precision data.• In 64-bit mode, produces object code that will run on any of the 64-bit PowerPC hardware platforms but not 32-bit-only platforms.• Defines the <code>_ARCH_PPC</code> macro. |
| 403 | <ul style="list-style-type: none">• Produces object code containing instructions that will run on the 403 hardware platform.• Defines the <code>_ARCH_PPC</code> and <code>_ARCH_403</code> macros. |
| 601 | <ul style="list-style-type: none">• Produces object code containing instructions that will run on the 601 hardware platform.• Defines the <code>_ARCH_601</code> macro. |

- 602
 - Produces object code containing instructions that will run on the 602 hardware platform.
 - Defines the `_ARCH_PPC` and `_ARCH_602` macros.
- 603
 - Produces object code containing instructions that will run on the 603 hardware platform.
 - Defines the `_ARCH_PPC`, `_ARCH_PPCGR`, and `_ARCH_603` macros.
- 604
 - Produces object code containing instructions that will run on the 604 hardware platform.
 - Defines the `_ARCH_PPC`, `_ARCH_PPCGR`, and `_ARCH_604` macros.
- pwr
 - Produces object code containing instructions that will run on any of the POWER, POWER2, and 601 hardware platforms .
 - Defines the `_ARCH_PWR` macro.
- pwr2
pwrX
 - Produces object code containing instructions that will run on the POWER2 hardware platforms, including pwr2s and p2sc.
 - Defines the `_ARCH_PWR` and `_ARCH_PWR2` macros.
- pwr2s
 - Produces object code containing instructions that will run on the pwr2s hardware platform.
 - Defines the `_ARCH_PWR`, `_ARCH_PWR2`, and `_ARCH_PWR2S` macros.
- p2sc
 - Produces object code containing instructions that will run on the p2sc hardware platform.
 - Defines the `_ARCH_PWR`, `_ARCH_PWR2`, and `_ARCH_P2SC` macros.
- pwr3
 - Produces object code containing instructions that will run on the POWER3 hardware platforms.
 - Defines the `_ARCH_PPC`, `_ARCH_PPCGR`, `_ARCH_PPCGRSQ`, and `_ARCH_PWR3` macros.
- pwr4
 - Produces object code containing instructions that will run on the POWER4 hardware platforms.
 - Defines the `_ARCH_PPC`, `_ARCH_PPCGR`, `_ARCH_PPCGRSQ`, and `_ARCH_PWR4` macros.
- PPCGR
 - In 32-bit mode, produces object code containing optional graphics instructions for PowerPC processors.
 - In 64-bit mode, produces object code containing optional graphics instructions that will run on 64-bit PowerPC hardware platforms but not on 32-bit-only platforms.
 - Defines the `_ARCH_PPC` and `_ARCH_PPCGR` macros.
- PPCGRSQ
 - In 32-bit mode, produces object code containing optional graphics and sqrt instructions that will run on PowerPC processors.
 - In 64-bit mode, produces object code containing optional graphics instructions and sqrt that will run on 64-bit PowerPC hardware processors, but not on 32-bit-only processors.
 - Defines the `_ARCH_PPC`, `_ARCH_PPCGR`, and `_ARCH_PPCGRSQ` macros.
- rs64a
 - Produces object code that will run on rs64a processors.
 - Defines the `_ARCH_PPC` and `_ARCH_RS64A` macros.
- rs64b
 - Produces object code that will run on rs64b processors.
 - Defines the `_ARCH_PPC`, `_ARCH_PPCGR`, `_ARCH_PPCGRSQ`, and `_ARCH_RS64B` macros.
- rs64c
 - Produces object code that will run on rs64c processors.
 - Defines the `_ARCH_PPC`, `_ARCH_PPCGR`, `_ARCH_PPCGRSQ`, and `_ARCH_RS64C` macros.

See also “#pragma options” on page 325.

Default

The default setting of `-qarch` is `-qarch=com` unless the `OBJECT_MODE` environment variable is set to `64`.

Notes

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

Using `-qarch=ppc` or any ppc family architecture with `-qfloat=hssngl` or `-qfloat=hsflt` may produce incorrect results on rs64b or future systems.

You can use `-qarch=suboption` with `-qtune=suboption`. `-qarch=suboption` specifies the architecture for which the instructions are to be generated, and `-qtune=suboption` specifies the target platform for which the code is optimized.

Example

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

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

Related Tasks

“Specify Compiler Options for Architecture-Specific, 32- or 64-bit Compilation” on page 29

Related References

“Compiler Command Line Options” on page 61

“float” on page 131

“O, optimize” on page 215

“tune” on page 277

“Acceptable Compiler Mode and Processor Architecture Combinations” on page 373

assert

▶ C

Purpose

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

Syntax



where available aliasing options include:

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

See also “#pragma options” on page 325.

Notes

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

Related References

“Compiler Command Line Options” on page 61

“alias” on page 75

attr

C C++

Purpose

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

Syntax



where:

- qattr=full Reports all identifiers in the program.
- qattr Reports only those identifiers that are used.

See also “#pragma options” on page 325.

Notes

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

The -qnoprint option overrides this option.

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

Example

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

```
xlc myprogram.c -qxref -qattr=full
```

A typical cross-reference listing has the form:

Identifier name	Description of the item
xy	auto int in function adder
	0-59Y 0-36.12Z 0-48.12Z
	Function invocation
	Column number
	Line number
	File
	Function definition

Related References

- “Compiler Command Line Options” on page 61
- “print” on page 231
- “xref” on page 294
- “#pragma options” on page 325

B

> C > C++

Purpose

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

Syntax

►► -B prefix -t-program ►►

where *program* can be:

<i>program</i>	Description
a	Assembler
b	Compiler back end
c	Compiler front end
I	Interprocedural Analysis tool
l	linkage editor
p	compiler preprocessor

Notes

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

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

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

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

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

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

Example

To compile myprogram.c using a substitute x1C compiler in */lib/tmp/mine/* enter:

```
x1C myprogram.c -B/lib/tmp/mine/
```

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

```
x1C myprogram.c -B/lib/tmp/mine/ -tl
```

Related References

“Compiler Command Line Options” on page 61

“path” on page 225

b

► C ► C++

Purpose

Controls how shared objects are processed by the linkage editor.

Syntax

►► -b

dynamic
shared
static

 ►►►

where options are:

dynamic, shared	Causes the linker to process subsequent shared objects in dynamic mode. This is the default. In dynamic mode, shared objects are not statically included in the output file. Instead, the shared objects are listed in the loader section of the output file.
static	Causes the linker to process subsequent shared objects in static mode. In static mode, shared objects are statically linked in the output file.

Notes

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

For more information about this and other **ld** options, see the *AIX Commands Reference*.

Related References

“Compiler Command Line Options” on page 61

See also:

ld command in *Commands Reference, Volume 5: s through u*

bitfields

> C > C++

Purpose

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

Syntax

►► -q-bitfields= [unsigned] [signed] ◄◄

where options are:

signed	Bitfields are signed.
unsigned	Bitfields are unsigned.

Related References

“Compiler Command Line Options” on page 61

bmaxdata

► C ► C++

Purpose

This option sets the maximum size of the area shared by the static data (both initialized and uninitialized) and the heap to *size* bytes. This value is used by the system loader to set the soft ulimit.

The default setting is **-bmaxdata=0**.

Syntax

►► -bmaxdata=0
number►►

Notes

Valid values for *number* are 0 and multiples of 0x10000000 (0x10000000, 0x20000000, 0x30000000, ...). The maximum value allowed by the system is 0x80000000.

If the value of *size* is 0, a single 256MB (0x10000000 byte) data segment (segment 2) will be shared by the static data, the heap, and the stack. If the value is non-zero, a data area of the specified size (starting in segment 3) will be shared by the static data and the heap, while a separate 256 MB data segment (segment 2) will be used by the stack. So, the total data size when 0 is specified is 256MB, and the total size when 0x10000000 is specified is 512MB, with 256MB for the stack and 256MB for static data and the heap.

Related References

“Compiler Command Line Options” on page 61

brtl

> C > C++

Purpose

Enables run-time linking for the output file.

Syntax

▶▶ — -brtl —————▶▶

Notes

DCE thread libraries and heap debug libraries are not compatible with runtime linking. Do not specify the **-qbrtl** compiler option if you are invoking the compiler with **xlC_r4** or **xlC_r4**.

Run-time linking is the ability to resolve undefined and non-deferred symbols in shared modules after the program execution has already begun. It is a mechanism for providing run-time definitions (these function definitions are not available at link-time) and symbol rebinding capabilities. The *main* application must be built to enable run-time linking. You cannot simply link any module with the run-time linker.

To include runtime linking in your program, compile using the **-brtl** compiler option. This will add a reference to the runtime linker to your program, which will be called by your program's start-up code (`/lib/crt0.o`) when program execution begins. Shared object input files are listed as dependents in the program loader section in the same order as they are specified on the command line. When the program execution begins, the system loader loads these shared objects so their definitions are available to the runtime linker.

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

Related References

"Compiler Command Line Options" on page 61

"b" on page 89

"G" on page 140

Also, see the Shared Objects and Runtime Linking chapter in General Programming Concepts: Writing and Debugging Programs.

C

► C ► C++

Purpose

Preserves comments in preprocessed output.

Syntax

►► -C ◀◀

Notes

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

Example

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

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

Related References

“Compiler Command Line Options” on page 61

“E” on page 115

“P” on page 222

C

► C ► C++

Purpose

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

Syntax

►► -c ◀◀

Notes

The compiled source files are not sent to the linkage editor. The compiler creates an output object file, *file_name.o*, for each valid source file, *file_name.c* or *file_name.i*.

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

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

Example

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

```
x1C myprogram.c -c
```

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

```
x1C myprogram.c -c -o new.o
```

Related References

“Compiler Command Line Options” on page 61

“E” on page 115

“o” on page 219

“P” on page 222

“syntaxonly” on page 265

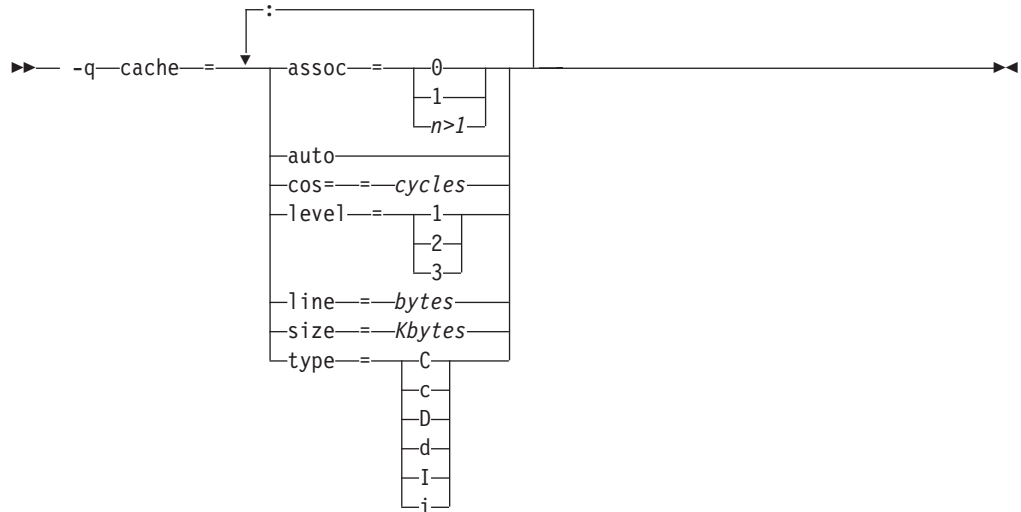
cache

► C ► C++

Purpose

The `-qcache` option specifies the cache configuration for a specific execution machine. If you know the type of execution system for a program, and that system has its instruction or data cache configured differently from the default case, use this option to specify the exact cache characteristics. The compiler uses this information to calculate the benefits of cache-related optimizations.

Syntax



where available cache options are:

<code>assoc=number</code>	Specifies the set associativity of the cache, where <i>number</i> is one of: 0 Direct-mapped cache 1 Fully associative cache N>1 n-way set associative cache
<code>auto</code>	Automatically detects the specific cache configuration of the compiling machine. This assumes that the execution environment will be the same as the compilation environment.
<code>cost=cycles</code>	Specifies the performance penalty resulting from a cache miss.
<code>level=level</code>	Specifies the level of cache affected, where <i>level</i> is one of: 1 Basic cache 2 Level-2 cache or, if there is no level-2 cache, the table lookaside buffer (TLB) 3 TLB If a machine has more than one level of cache, use a separate <code>-qcache</code> option.
<code>line=bytes</code>	Specifies the line size of the cache.
<code>size=Kbytes</code>	Specifies the total size of the cache.

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

C or c Combined data and instruction cache

D or d Data cache

I or i Instruction cache

Notes

If you specify the wrong values for the cache configuration or run the program on a machine with a different configuration, the program will work correctly but may be slightly slower.

You must specify **-O4**, **-O5**, or **-qipa** with the **-qcache** option.

Use the following guidelines when specifying **-qcache** suboptions:

- Specify information for as many configuration parameters as possible.
- If the target execution system has more than one level of cache, use a separate **-qcache** option to describe each cache level.
- If you are unsure of the exact size of the cache(s) on the target execution machine, specify an estimated cache size on the small side. It is better to leave some cache memory unused than it is to experience cache misses or page faults from specifying a cache size larger than actually present.
- The data cache has a greater effect on program performance than the instruction cache. If you have limited time available to experiment with different cache configurations, determine the optimal configuration specifications for the data cache first.
- If you specify the wrong values for the cache configuration, or run the program on a machine with a different configuration, program performance may degrade but program output will still be as expected.
- The **-O4** and **-O5** optimization options automatically select the cache characteristics of the compiling machine. If you specify the **-qcache** option together with the **-O4** or **-O5** options, the option specified last takes precedence.

Example

To tune performance for a system with a combined instruction and data level-1 cache, where cache is 2-way associative, 8 KB in size and has 64-byte cache lines, enter:

```
xlc -O4 -qcache=type=c:level=1:size=8;line=64;assoc=2 file.C
```

Related References

“Compiler Command Line Options” on page 61

“ipa” on page 163

“O, optimize” on page 215

chars

► C ► C++

Purpose

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

Syntax

►► -qchars=signed | unsigned ►►

See also “#pragma chars” on page 301 and “#pragma options” on page 325.

Notes

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

```
#pragma options chars=sign_type
```

```
#pragma chars (sign_type)
```

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

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

Example

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

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

Related References

“Compiler Command Line Options” on page 61

“#pragma chars” on page 301

“#pragma options” on page 325

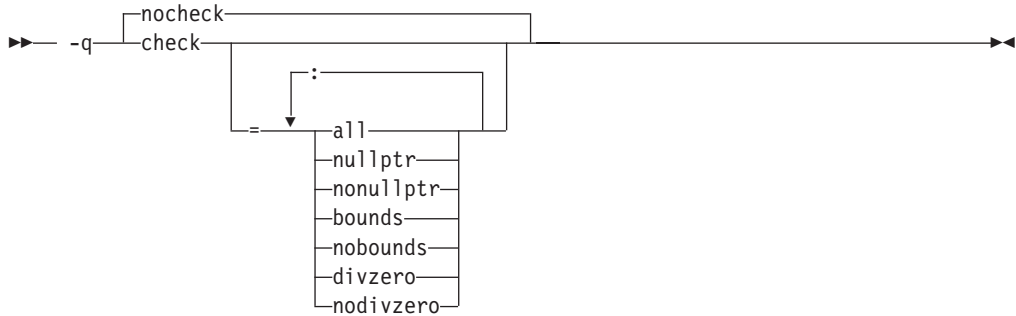
check

C C++

Purpose

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

Syntax



where:

all Switches on all the following suboptions. You can use the **all** option along with the **no...** form of one or more of the other options as a filter.

For example, using:

```
xlc myprogram.c -qcheck=all:nonnull
```

provides checking for everything except for addresses contained in pointer variables used to reference storage.

If you use **all** with the **no...** form of the options, **all** should be the first suboption.

NULLptr | NONULLptr Performs runtime checking of addresses contained in pointer variables used to reference storage. The address is checked at the point of use; a trap will occur if the value is less than 512.

bounds | nobounds Performs runtime checking of addresses when subscripting within an object of known size. The index is checked to ensure that it will result in an address that lies within the bounds of the object's storage. A trap will occur if the address does not lie within the bounds of the object.

DIVzero | NODIVzero Performs runtime checking of integer division. A trap will occur if an attempt is made to divide by zero.

See also “#pragma options” on page 325.

Notes

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

Using the **-qcheck** option without any suboptions turns all the suboptions on.

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

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

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

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

Examples

1.

For **-qcheck=null:bounds**:

```
void func1(int* p) {
    *p = 42;          /* Traps if p is a null pointer */
}

void func2(int i) {
    int array[10];
    array[i] = 42;   /* Traps if i is outside range 0 - 9 */
}
```

2.

For **-qcheck=divzero**:

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

Related References

“Compiler Command Line Options” on page 61

“#pragma options” on page 325

cinc

C++

Purpose

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

Syntax



where:

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

Notes

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

Related References

"Compiler Command Line Options" on page 61

compact

► C ► C++

Purpose

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

Syntax

►► -q compact nocompact ◀◀

See also “#pragma options” on page 325.

Notes

Code size is reduced by inhibiting optimizations that replicate or expand code inline. Execution time may increase.

Example

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

```
x1C myprogram.c -qcompact
```

Related References

“Compiler Command Line Options” on page 61

“#pragma options” on page 325

cplusplus



Purpose

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

Syntax



Default

The default setting varies according to the **langlvl** compiler option setting.

- If **langlvl** is set to **stdc99** or **extc99**, **cplusplus** is implicitly selected. You can override this implicit selection by specifying **-qlanglvl=stdc99 -qnocplusplus** or **-qlanglvl=extc99 -qnocplusplus**.
- Otherwise, the default setting is **nocplusplus**.

Notes

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

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

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

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

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

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

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

The preprocessor handles all comments in the following ways:

- If the **-C** option is *not* specified, all comments are removed and replaced by a single blank.
- If the **-C** option *is* specified, comments are output unless they appear on a preprocessor directive or in a macro argument.
- If **-E** is specified, continuation sequences are recognized in all comments and are output
- If **-P** is specified, comments are recognized and stripped from the output, forming concatenated output lines.

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

Examples

1. Example of C++ Comments

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

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

2. Preprocessor Output Example 1

For the following source code fragment:

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

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

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

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

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

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

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

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

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

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

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

3. Preprocessor Output Example 2 - Directive Line

For the following source code fragment:

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

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

```
int a;
int b;

int c;
```

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

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

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

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

int c;
```

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

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

4. Preprocessor Output Example 3 - Macro Function Argument

For the following source code fragment:

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

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

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

int d;
```

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

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

The output for the -E option is:

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

int d;
```

The output for the -E-C option is:

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

5. Compile Example

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

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

Related References

"Compiler Command Line Options" on page 61

"C" on page 93

"E" on page 115

"langlvl" on page 175

"P" on page 222

D

> C > C++

Purpose

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

Syntax

► -D *name* [= *definition*] ►

Notes

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

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

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

To aid in program portability and standards compliance, the AIX Version 4 Operating System provides several header files that define macro names you can set with the **-D** option. You can find most of these header files either in the **/usr/include** directory or in the **/usr/include/sys** directory. See "Header Files Overview" in the *AIX Version 4 Files Reference* for more information.

The configuration file uses the **-D** option to specify the following predefined macros:

Macro name	Applies to AIX v5.1	Applies to AIX v4.3
AIX	✓	✓
AIX32	✓	✓
AIX41	✓	✓
AIX43	✓	✓
AIX50	✓	
AIX51	✓	
IBMR2	✓	✓
POWER	✓	✓
ANSI_C_SOURCE	✓	✓

To ensure that the correct macros for your source file are defined, use the **-D** option with the appropriate macro name. If your source file includes the **/usr/include/sys/stat.h** header file, you must compile with the option **-D_POSIX_SOURCE** to pick up the correct definitions for that file.

If your source file includes the **/usr/include/standards.h** header file, **_ANSI_C_SOURCE**, **_XOPEN_SOURCE**, and **_POSIX_SOURCE** are defined if you have not defined any of them.

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

Examples

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

```
xlc myprogram.c -D_LARGE_FILES -qlonglong
```

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

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

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

Related References

"Compiler Command Line Options" on page 61

"U" on page 281

See also:

Header Files command in the Files Reference


dataimported

> C > C++

Purpose

Marks data as imported.

Syntax

►► -q-dataimported 

Notes

Imported variables are dynamically bound with a shared portion of a library. **-qdataimported** changes the default to assume that all variables are imported. Specifying **-qdataimported=names** marks the named variables as imported, where *names* is a list of identifiers separated by colons (:). The default is not changed.

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

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

Related References

"Compiler Command Line Options" on page 61

"datalocal" on page 109

datalocal

► C ► C++

Purpose

Marks data as local.

Syntax



Notes

Local variables are statically bound with the functions that use them. **-qdatalocal** changes the default to assume that all variables are local. Specifying **-qdatalocal=names** marks the named variables as local, where *names* is a list of identifiers separated by colons (:). The default is not changed. Performance may decrease if an imported variable is assumed to be local.

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

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

Related References

“Compiler Command Line Options” on page 61

“dataimported” on page 108

dbxextra

► C

Purpose

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

Syntax

► — -q — nodbxextra — dbxextra —

See also “#pragma options” on page 325.

Notes

Use this option with the **-g** option to produce additional debugging information for use with the IBM Distributed Debugger.

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

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

Example

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

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

Related References

“Compiler Command Line Options” on page 61

“g” on page 141

“#pragma options” on page 325

Related References

"Compiler Command Line Options" on page 61

"langlvl" on page 175

"#pragma options" on page 325

dollar

► C ► C++

Purpose

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

Syntax

► — -q — nodollar — dollar — ◀◀

See also “#pragma options” on page 325.

Example

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

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

Related References

“Compiler Command Line Options” on page 61

“#pragma options” on page 325

dpcl

> C > C++

Purpose

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

Syntax

► — -q — nodpcl
dpcl —————►

Notes

When you specify the **-qdpcl** option, the compiler emits symbols to define blocks of code in a program. You can then use tools that use the DPCL interface to examine performance information such as memory usage for object files that you have compiled with this option.

You must also specify the **-g** option when you specify **-qdpcl**.

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

Related References

"Compiler Command Line Options" on page 61

"dpcl"

"g" on page 141

"ipa" on page 163

"smp" on page 252

Example

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

```
x1C myprogram.c -E
```

If myprogram.c has a code fragment such as:

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

int c ;

/* Another comment */
c = SUM(a, /* Comment in a macro function argument*/
b) ;
```

the output will be:

```
#line 2 "myprogram.c"
int a;
#line 5
int b;

int c;

c =
(a + b);
```

Related References

“Compiler Command Line Options” on page 61

“E” on page 115

“M” on page 198

“o” on page 219

“P” on page 222

“syntaxonly” on page 265

e

► C ► C++

Purpose

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

Syntax

►► — *-e—name* —————►►

Related References

“Compiler Command Line Options” on page 61

“mkshrobj” on page 210

eh

C++

Purpose

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

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

If your program uses C++ exception handling, the program behaviour is undefined if **-qnoeh** is specified

Syntax

►► -q eh noeh ◄◄

Related References

“Compiler Command Line Options” on page 61

enum

C C++

Purpose

Specifies the amount of storage occupied by enumerations.

Syntax



where valid **enum** settings are:

- `-qenum=small` Specifies that enumerations occupy a minimum amount of storage: either 1, 2, or 4 bytes of storage, depending on the range of the **enum** constants. In 64-bit compilation mode, the enumerations can also use 8 bytes of storage.
- `-qenum=int` Specifies that enumerations occupy 4 bytes of storage and are represented by **int**.
- `-qenum=1` Specifies that enumerations occupy 1 byte of storage.
- `-qenum=2` Specifies that enumerations occupy 2 bytes of storage.
- `-qenum=4` Specifies that enumerations occupy 4 bytes of storage.
- `-qenum=8` Valid only in 64-bit compiler mode. Specifies that enumerations occupy 8 bytes of storage.
- `-qenum=intlong` **C++** Valid only in 64-bit compiler mode. Specifies that enumerations occupy 8 bytes of storage and are represented by **long**, if `-q64` is specified and the range of the **enum** constants exceed the limit for **int**. Otherwise, the enumerations occupy 4 bytes of storage and are represented by **int**.

See also “`#pragma enum`” on page 305 and “`#pragma options`” on page 325.

Notes

The **enum constants** are always of type **int**, except for the following cases:

- If `-q64` is not specified, and if the range of these constants is beyond the range of **int**, **enum constants** will have type **unsigned int** and be 4 bytes long.
- If `-q64` is specified, and if the range of these constants is beyond the range of **int**, **enum constants** will have type **long** and be 8 bytes long.

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

The `-qenum=1|2|4|8` options allocate a specific amount of storage to an **enum variable**. If the specified storage size is smaller than that required by the range of **enum variables**, the requested size is kept but a warning is issued. For example:

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

For each **#pragma options enum=** directive that you put in a source file, it is good practice to have a corresponding **#pragma options enum=reset** before the end of that file. This is the only way to prevent one file from potentially changing the **enum=** setting of another file that **#includes** it. The **#pragma enum()** directive can be instead of **#pragma options enum=**. The two pragmas are interchangeable.

The tables below show the priority for selecting a predefined type. They also shows the the predefined type, the maximum range of **enum** constants for the corresponding predefined type, and the amount of storage that is required for that predefined type (that is, the value that the **sizeof** operator would yield when applied to the minimum-sized **enum**).

32-bit Compilation Mode

Range	enum=int		enum=small		enum=1		enum=2		enum=4	
	variable	constant	variable	constant	variable	constant	variable	constant	variable	constant
0 .. 127	int	int	unsigned char	int	signed char	int	short	int	int	int
-128 .. 127	int	int	signed char	int	signed char	int	short	int	int	int
0 .. 255	int	int	unsigned char	int	unsigned char	int	short	int	int	int
0 .. 32767	int	int	unsigned short	int	unsigned short ¹	int	short	int	int	int
-32768 .. 32767	int	int	short	int	short ¹	int	short	int	int	int
0 .. 65535	int	int	unsigned short	int	unsigned short ¹	int	unsigned short	int	int	int
0 .. 2147483647	int	int	unsigned int	unsigned int	unsigned int ¹	int	unsigned int ¹	int	int	int
-(2147483647+1) .. 2147483647	int	int	int	int	int ¹	int	int ¹	int	int	int
0 .. 4294967295	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int ¹	unsigned int	unsigned int ¹	unsigned int	unsigned int	unsigned int

Note:

1. These enumerations are too large to the particular **enum=1|2|4** option. The size of the enum is increased to hold the entire range of values. It is recommended that you change the enum option to match the size of the enum required.

64-bit Compilation Mode														
Range	enum=int		C++ enum=intlong		enum=small		enum=1		enum=2		enum=4		enum=8	
	var	const	var	const	var	const	var	const	var	const	var	const	var	const
0..127	int	int	int	int	unsigned char	int	signed char	int	short	int	int	int	long	const
-128..127	int	int	int	int	signed char	int	signed char	int	short	int	int	int	long	const
0..255	int	int	int	int	unsigned char	int	unsigned char	int	short	int	int	int	long	const
0..32767	int	int	int	int	unsigned short	int	unsigned short ¹	int	short	int	int	int	long	const
-32768..32767	int	int	int	int	short	int	short ¹	int	short	int	int	int	long	const
0..65535	int	int	int	int	unsigned short	int	unsigned short ¹	int	unsigned short	int	int	int	long	const
0..2147483647	int	int	int	int	unsigned int	int	int ¹	int	int ¹	int	int	int	long	const
-(2147483647+1) ..2147483647	int	int	int	int	int	int	int ¹	int	int ¹	int	int	int	long	const
0..4294967295	unsigned int	unsigned int	unsigned int	unsigned int	unsigned int	int	unsigned int ¹	int	unsigned int ¹	int	unsigned int	int	long	const
0..(2 ⁶³ -1)	ERR ²	ERR ²	long	long	unsigned long	long	long ¹	long	long ¹	long	long ¹	long	long	const
-2 ⁶³ ..(2 ⁶³ -1)	ERR ²	ERR ²	long	long	long	long	long ¹	long	long ¹	long	long ¹	long	long	const
0..2 ⁶⁴	ERR ²	ERR ²	unsigned long	unsigned long	unsigned long	long	unsigned long ¹	long	unsigned long ¹	long	unsigned long ¹	long	unsigned long	const

Notes:

1. These enumerations are too large to the particular **enum=1|2|4** option. The size of the enum is increased to hold the entire range of values. It is recommended that you change the enum option to match the size of the enum required.
2. These enumerations are too large for the enum=int option. It is recommended that you change reduce the range of the values of the enumerations or change the enum option to **enum=intlong**.

The following are invalid enumerations or invalid usage of **#pragma options enum=**:

- You cannot change the storage allocation of an enum using a **#pragma options enum=** within the declaration of an enum. The following code segment generates a warning and the second occurrence of the **enum** option is ignored:

```
#pragma options enum=small
enum e_tag {
    a,
    b,
    #pragma options enum=int /* error: cannot be within a declaration */
    c
} e_var;
#pragma options enum=reset /* second reset isn't required */
```

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

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

- The **enum** constant range does not fit within the range of an **unsigned int**.

```
#pragma options enum=small
enum e_tag { a=0,
             b=4294967296 /* error: larger than maximum int */
             } e_var;
#pragma options enum=reset
```

A **-qenum=reset** option corresponding to the **#pragma options enum=reset** directive does not exist. Attempting to use **-qenum=reset** generates a warning message and the option is ignored.

Examples

1. One typical use for the **reset** suboption is to reset the enumeration size set at the end of an include file that specifies an enumeration storage different from the default in the main file. For example, the following include file, `small_enum.h`, declares various minimum-sized enumerations, then resets the specification at the end of the include file to the last value on the option stack:

```
#ifndef small_enum_h
#define small_enum_h 1
/*
 * File small_enum.h
 * This enum must fit within an unsigned char type
 */
#pragma options enum=small
enum e_tag {a, b=255};
enum e_tag u_char_e_var; /* occupies 1 byte of storage */

/* Reset the enumeration size to whatever it was before */
#pragma options enum=reset
#endif
```

The following source file, `int_file.c`, includes `small_enum.h`:

```
/*
 * File int_file.c
 * Defines 4 byte enums
 */
#pragma options enum=int
enum testing {ONE, TWO, THREE};
enum testing test_enum;
```

```

/* various minimum-sized enums are declared */
#include "small_enum.h"

/* return to int-sized enums. small_enum.h has reset the
 * enum size
 */
enum sushi {CALIF_ROLL, SALMON_ROLL, TUNA, SQUID, UNI};
enum sushi first_order = UNI;

```

The enumerations **test_enum** and **test_order** both occupy 4 bytes of storage and are of type **int**. The variable **u_char_e_var** defined in `small_enum.h` occupies 1 byte of storage and is represented by an **unsigned char** data type.

2. If the following C fragment is compiled with the **enum=small** option:

```
enum e_tag {a, b, c} e_var;
```

the range of enum constants is 0 through 2. This range falls within all of the ranges described in the table above. Based on priority, the compiler uses predefined type **unsigned char**.

3. If the following C code fragment is compiled with the **enum=small** option:

```
enum e_tag {a=-129, b, c} e_var;
```

the range of enum constants is -129 through -127. This range only falls within the ranges of **short (signed short)** and **int (signed int)**. Because **short (signed short)** smaller, it will be used to represent the **enum**.

4. If you compile a file `myprogram.c` using the command:

```
x1C myprogram.c -qenum=small
```

assuming file `myprogram.c` does not contain **#pragma options=int** statements, all **enum** variables within your source file will occupy the minimum amount of storage.

5. If you compile a file `yourfile.c` that contains the following lines:

```
enum testing {ONE, TWO, THREE};
enum testing test_enum;

#pragma options enum=small
enum sushi {CALIF_ROLL, SALMON_ROLL, TUNA, SQUID, UNI};
enum sushi first_order = UNI;

#pragma options enum=int
enum music {ROCK, JAZZ, NEW_WAVE, CLASSICAL};
enum music listening_type;
```

using the command:

```
x1C yourfile.c
```

only the enum variable **first_order** will be minimum-sized (that is, enum variable **first_order** will only occupy 1 byte of storage). The other two enum variables **test_enum** and **listening_type** will be of type **int** and occupy 4 bytes of storage.

Related References

“Compiler Command Line Options” on page 61

“#pragma enum” on page 305

“#pragma options” on page 325

expfile

► C ► C++

Purpose

Saves all exported symbols in a designated file.

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

Syntax

►► — -q—expfile—=*filename*—————▶▶

Related References

“Compiler Command Line Options” on page 61

“mkshrobj” on page 210

extchk

> C > C++

Purpose

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

Syntax

►► -q noextchk
extchk ◄◄

See also “#pragma options” on page 325.

Notes

-q**extchk** checks for consistency at compile time and detects mismatches across compilation units at link time.

-q**extchk** does not perform type checking on functions or objects that contain references to incomplete types.

Example

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

```
x1C myprogram.c -qextchk
```

Related References

“Compiler Command Line Options” on page 61

“#pragma options” on page 325

F

► C ► C++

Purpose

Names an alternative configuration file (.cfg) for x1C.

Syntax

```
► -F config_file [ :—stanza ]
```

where suboptions are:

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

Notes

The default is a configuration file supplied at installation time called (/etc/vac.cfg). Any file names or stanzas that you specify on the command line or within your source file override the defaults specified in the /etc/vac.cfg configuration file.

For information regarding the contents of the configuration file, refer to “Specify Compiler Options in a Configuration File” on page 27.

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

Example

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

```
x1C myprogram.c -F/usr/tmp/myvac.cfg:x1C
```

Related Tasks

“Specify Compiler Options in a Configuration File” on page 27

Related References

“Compiler Command Line Options” on page 61

“B” on page 88

“t” on page 266

“W” on page 290

f

> C > C++

Purpose

Names a file to store a list of object files for `x1C` to pass to the linker.

Syntax

▶▶ `-f—filelistname` ▶▶

Notes

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

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

Example

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

```
x1C -f/usr/tmp/myobjlistfile
```

Related References

“Compiler Command Line Options” on page 61

fdpr

► C ► C++

Purpose

Collects information about your program for use with the AIX **fdpr** (Feedback Directed Program Restructuring) performance-tuning utility.

Syntax

► — -q — nofdpr fdpr —————►

Notes

You should compile your program with **-qfdpr** before optimizing it with the **fdpr** performance-tuning utility. Optimization data is stored in the object file.

For more information on using the **fdpr** performance-tuning utility, refer to the *AIX Version 4 Commands Reference* or enter the command:

```
man fdpr
```

Example

To compile `myprogram.c` so it include data required by the **fdpr** utility, enter:

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

Related References

“Compiler Command Line Options” on page 61

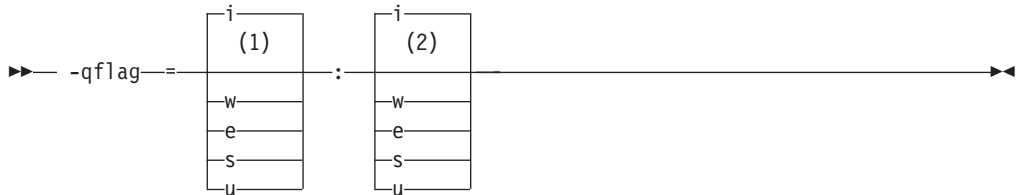
flag

C C++

Purpose

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

Syntax



Notes:

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

where message severity levels are:

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

See also “#pragma options” on page 325.

Notes

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

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

Example

To compile myprogram.c so that the listing shows all messages that were generated and your workstation displays only error and higher messages (with their associated information messages to aid in fixing the errors), enter:

```
xlc myprogram.c -qflag=I:E
```

Related References

“Compiler Command Line Options” on page 61

“info” on page 154

“#pragma options” on page 325

“Compiler Messages” on page 379

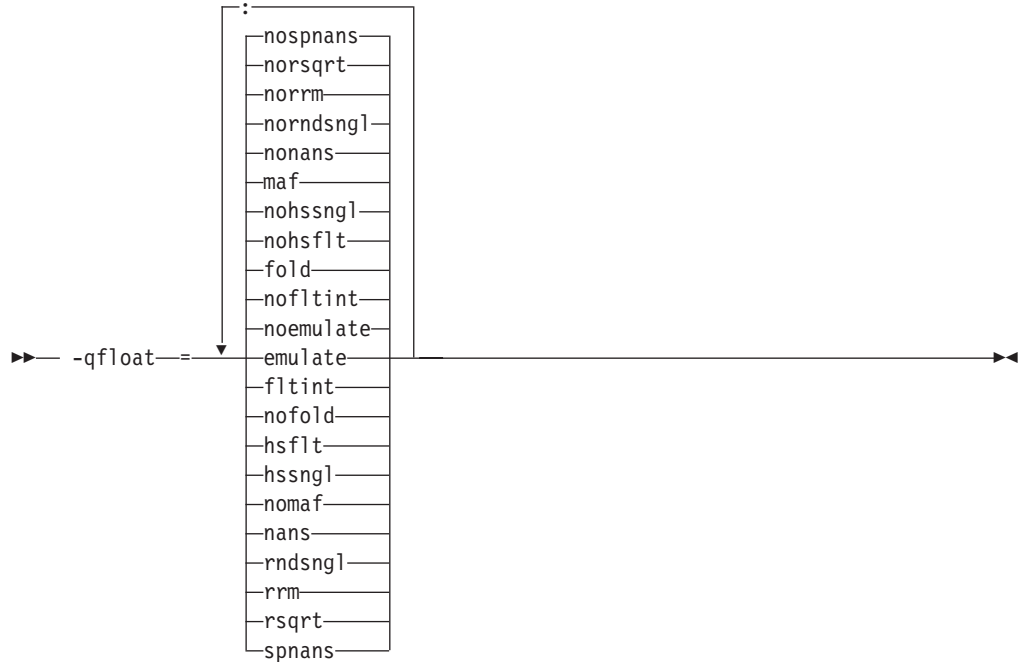
float

C C++

Purpose

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

Syntax



Option selections are described in the **Notes** section below. See also “#pragma options” on page 325.

Notes

Using the `float` option may produce results that are not precisely the same as the default. Incorrect results may be produced if not all required conditions are met. For these reasons, you should only use this option if you are experienced with floating-point calculations involving IEEE floating-point values and can properly assess the possibility of introducing errors in your program.

The `float` option has the following suboptions.

<code>-qfloat=emulate</code> <code>-qfloat=noemulate</code>	<p>Emulates the floating-point instructions omitted by the PowerPC 403 processor. The default is float=noemulate.</p> <p>To emulate PowerPC 403 processor floating-point instructions, use -qfloat=emulate. Function calls are emitted in place of PowerPC 403 floating-point instructions. Use this option only in a single-threaded, stand-alone environment targeting the PowerPC 403 processor.</p> <p>Do not use -qfloat=emulate with any of the following:</p> <ul style="list-style-type: none">• -qarch=pwr, -qarch=pwr2, -qarch=pwrx• -qlongdouble, -qldb128• xlC128 or xlC128 compiler invocation commands
----------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<p>-qfloat=fltint -qfloat=nofltint</p>	<p>Speeds up floating-point-to-integer conversions by using faster inline code that does not check for overflows. The default is float=nofltint, which checks floating-point-to-integer conversions for out-of-range values.</p> <p>This suboption must only be used with an optimization option.</p> <ul style="list-style-type: none"> • For -O2, the default is -qfloat=nofltint. • For -O3, the default is -qfloat=fltint. <p>To include range checking in floating-point-to-integer conversions with the -O3 option, specify -qfloat=nofltint.</p> <ul style="list-style-type: none"> • -qnostrict sets -qfloat=fltint <p>Changing the optimization level will not change the setting of the fltint suboption if fltint has already been specified.</p> <p>This option is ignored unless -qarch=pwr or, in 32-bit mode, -qarch=com. For PWR2 and PPC family architectures, faster inline code is used that correctly handles out-of-range values.</p> <p>If the -qstrict -qnostrict and -qfloat= options conflict, the last setting is used.</p>
<p>-qfloat=fold -qfloat=nofold</p>	<p>Specifies that constant floating-point expressions are to be evaluated at compile time rather than at run time.</p> <p>The -qfloat=fold option replaces the obsolete -qfold option. Use -qfloat=fold in your new applications.</p>
<p>-qfloat=hsflt -qfloat=nohsflt</p>	<p>Speeds up calculations by truncating instead of rounding computed values to single precision before storing and on conversions from floating point to integer. The nohsflt suboption specifies that single-precision expressions are rounded after expression evaluation and that floating-point-to-integer conversions are to be checked for out-of-range values.</p> <p>The hsflt suboption overrides the rndsngl, nans, and spnans suboptions.</p> <p>Note: The hsflt suboption is for specific applications in which floating-point computations have known characteristics. Using this option when you are compiling other application programs can produce incorrect results without warning.</p> <p>The -qfloat=hsflt option replaces the obsolete -qhsflt option. Use -qfloat=hsflt in your new applications.</p> <p>This option has little effect unless the -qarch option is set to pwr, pwr2, pwrx, pwr2s or, in 32-bit mode, com. For PPC family architectures, all single-precision (float) operations are rounded and the option only affects double-precision (double) expressions cast to single-precision (float).</p> <p>Using this option with -qfloat=rndsngl or -q64 or -qarch=ppc or any PPC family architecture may produce incorrect results on rs64b or future systems.</p>

<p>-qfloat=hssngl -qfloat=nohssngl</p>	<p>Specifies that single-precision expressions are rounded only when the results are stored into float memory locations. nohssngl specifies that single-precision expressions are rounded after expression evaluation. Using hssngl can improve runtime performance but is safer than using -qfloat=hsflt.</p> <p>The -qfloat=hssngl option replaces the obsolete -qhssngl option. Use -qfloat=hssngl in your new applications.</p> <p>This suboption has little effect unless the -qarch option is set to pwr, pwr2, pwrx, pwr2s or, in 32-bit mode, com. For PPC family architectures, all single-precision (float) operations are rounded and the option only affects double-precision (double) expressions cast to single-precision (float) and used in an assignment operator for which a store instruction is generated.</p> <p>Using this suboption with -qfloat=rndsngl or -q64 or -qarch=ppc or any PPC family architecture may produce incorrect results on rs64b or future systems.</p>
<p>-qfloat=maf -qfloat=nomaf</p>	<p>Makes floating-point calculations faster and more accurate by using floating-point multiply-add instructions where appropriate. The results may not be exactly equivalent to those from similar calculations performed at compile time or on other types of computers. This option may affect the precision of floating-point intermediate results.</p> <p>The -qfloat=maf option replaces the obsolete -qmaf option. Use -qfloat=maf in your new applications.</p>
<p>-qfloat=nans -qfloat=nonans</p>	<p>Generates extra instructions to detect signalling NaN (Not-a-Number) when converting from single precision to double precision at run time. The option nonans specifies that this conversion need not be detected. -qfloat=nans is required for full compliance to the IEEE 754 standard.</p> <p>The hsflt option overrides the nans option.</p> <p>When used with the -qflttrap or -qflttrap=invalid option, the compiler detects invalid operation exceptions in comparison operations that occur when one of the operands is a signalling NaN.</p> <p>The -qfloat=nans option replaces the obsolete -qfloat=spnans option and the -qspnans option. Use -qfloat=nans in your new applications.</p>
<p>-qfloat=rndsngl -qfloat=norndsngl</p>	<p>Specifies that the result of each single-precision (float) operation is to be rounded to single precision. -qfloat=norndsngl specifies that rounding to single-precision happens only after full expressions have been evaluated. Using this option may sacrifice speed for consistency with results from similar calculations on other types of computers.</p> <p>The hsflt suboption overrides the rndsngl option.</p> <p>This suboption has no effect unless the -qarch option is set to pwr, pwr2, pwrx, pwr2s or, in 32-bit mode, com. For PPC family architectures, all single-precision (float) operations are rounded.</p> <p>Using this option with -qfloat=hssngl or -qfloat=hsflt may produce incorrect results on rs64b or future systems.</p> <p>The -qfloat=rndsngl option replaces the obsolete -qrndsngl option. Use -qfloat=rndsngl in your new applications.</p>

<p><code>-qfloat=rrm</code> <code>-qfloat=norm</code></p>	<p>Prevents floating-point optimizations that are incompatible with runtime rounding to plus and minus infinity modes. Informs the compiler that the floating-point rounding mode may change at run time or that the floating-point rounding mode is not <i>round to nearest</i> at run time.</p> <p><code>-qfloat=rrm</code> must be specified if the Floating Point Status and Control register is changed at run time (as well as for initializing exception trapping).</p> <p>The <code>-qfloat=rrm</code> option replaces the obsolete <code>-qrrm</code> option. Use <code>-qfloat=rrm</code> in your new applications.</p>
<p><code>-qfloat=rsqrt</code> <code>-qfloat=norsqrt</code></p>	<p>Specifies whether a sequence of code that involves division by the result of a square root can be replaced by calculating the reciprocal of the square root and multiplying. Allowing this replacement produces code that runs faster.</p> <ul style="list-style-type: none"> • For <code>-O2</code>, the default is <code>-qfloat=norsqrt</code>. • For <code>-O3</code>, the default is <code>-qfloat=rsqrt</code>. Use <code>-qfloat=norsqrt</code> to override this default. • <code>-qnostrict</code> sets <code>-qfloat=rsqrt</code>. (Note that <code>-qfloat=rsqrt</code> means that <code>errno</code> will <i>not</i> be set for any <code>sqrt</code> function calls.) • <code>-qfloat=rsqrt</code> has no effect when <code>-qarch=pwr2</code> is also specified. • <code>-qfloat=rsqrt</code> has no effect unless <code>-qignerrno</code> is also specified. <p>Changing the optimization level will not change the setting of the <code>rsqrt</code> option if <code>rsqrt</code> has already been specified. If the <code>-qstrict</code> <code>-qnostrict</code> and <code>-qfloat=</code> options conflict, the last setting is used.</p>
<p><code>-qfloat=spnans</code> <code>-qfloat=nospnans</code></p>	<p>Generates extra instructions to detect signalling NaN on conversion from single precision to double precision. The option <code>nospnans</code> specifies that this conversion need not be detected.</p> <p>The <code>hsflt</code> suboption overrides the <code>spnans</code> suboption.</p> <p>The <code>-qfloat=nans</code> option replaces the obsolete <code>-qfloat=spnans</code> and <code>-qspnans</code> options. Use <code>-qfloat=nans</code> in your new applications.</p>

Example

To compile `myprogram.c` so that range checking occurs and multiply-add instructions are not generated, enter:

```
x1C myprogram.c -qfloat=fltint:nomaf
```

Related References

“Compiler Command Line Options” on page 61

“arch” on page 83

“float” on page 131

“flttrap” on page 135

“ldbl128, longdouble” on page 190

“rrm” on page 246

“strict” on page 261

“#pragma options” on page 325

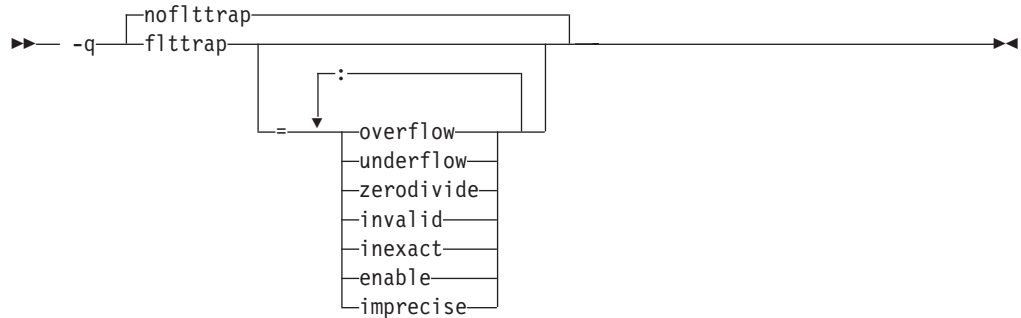
fltrap

C C++

Purpose

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

Syntax



where suboptions do the following:

Overflow	Generates code to detect and trap floating-point overflow.
UNDerflow	Generates code to detect and trap floating-point underflow.
ZERODivide	Generates code to detect and trap floating-point division by zero.
INValid	Generates code to detect and trap floating-point invalid operation exceptions.
INEXact	Generates code to detect and trap floating-point inexact exceptions.
ENable	Enables the specified exceptions in the prologue of the main program. This suboption is required if you want to turn on exception trapping without modifying the source code.
IMPrecise	Generates code for imprecise detection of the specified exceptions. If an exception occurs, it is detected, but the exact location of the exception is not determined.

See also “#pragma options” on page 325.

Notes

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

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

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

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

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

- with **-O**:

- 1/0 generates a **div0** exception and has a result of infinity
- 0/0 generates an invalid operation
- with **-O3**:
 - 1/0 generates a **div0** exception and has a result of infinity
 - 0/0 returns zero multiplied by the result of the previous division.

Example

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

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

Related References

“Compiler Command Line Options” on page 61

“float” on page 131

“O, optimize” on page 215

fold

► C ► C++

Purpose

Specifies that constant floating-point expressions are to be evaluated at compile time.

Syntax

►► -q fold nofold ◀◀

See also “#pragma options” on page 325.

Notes

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

Related References

“Compiler Command Line Options” on page 61

“float” on page 131

“#pragma options” on page 325

fullpath

> C > C++

Purpose

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

Syntax

► — -q — nofullpath — fullpath — ►

Notes

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

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

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

Related References

"Compiler Command Line Options" on page 61

"g" on page 141

funcsect

► C ► C++

Purpose

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

Syntax

► -q nofuncsect funcsect ◀

Notes

Using multiple csects increases the size of the object file, but often reduces the size of the final executable by allowing the linkage editor to remove functions that are not called or that have been inlined by the optimizer at all places they are called. If the file contains initialized static data or the pragma statement

```
#pragma comment copyright
```

some functions will be one machine word larger.

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

Related References

“Compiler Command Line Options” on page 61

“twolink” on page 279

G

C C++

Purpose

Tells the linkage editor to create a shared object enabled for runtime linking.

Syntax

►► -G ◀◀

Notes

The compiler will automatically export all global symbols from the shared object unless you specify which symbols to export by using **-bE:**, **-bexport:**, **-bexpall** or **-bnoexpall**.

If you use **-G** to create a shared library, the compiler will:

1. If the user doesn't specify **-bE:**, **-bexport:**, **-bexpall** or **-bnoexpall**, create an export list containing all global symbols using the CreateExportList script. You can specify another script with the **-tE/-B** or **-qpath=E:** options.
2. If CreateExportList was used to create the export list and **-qexpfile** was specified, the export list is saved.
3. Calls the linker with the appropriate options and object files to build a shared object.

This is a linkage editor (**ld**) option. Refer to your operating system documentation for a description of **ld** command usage and syntax.

Related References

"Compiler Command Line Options" on page 61

"B" on page 88

"b" on page 89

"brtl" on page 92

"expfile" on page 125

"path" on page 225

"t" on page 266

Also, on the Web see:

Shared Objects and Runtime Linking chapter in General Programming Concepts:
Writing and Debugging Programs

ld Command section in Commands Reference, Volume 3: i through m

g

> C > C++

Purpose

Generates debugging information used by tools such as the IBM Distributed Debugger.

Syntax

►► -g ◀◀

Notes

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

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

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

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

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

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

Example

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

```
x1C myprogram.c -o testing -g
```

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

```
x1C myprogram.c -o testing_all -g -qdbxextra
```

Related References

“Compiler Command Line Options” on page 61

“dbxextra” on page 110

“fullpath” on page 138

“linedebug” on page 192

“O, optimize” on page 215

“Q” on page 236

genproto

► C

Purpose

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

Syntax



Notes

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

Example

For the following function, foo.c:

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

specifying

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

produces

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

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

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

produces

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

In this case the parameter names are kept.

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

Related References

“Compiler Command Line Options” on page 61

halt

C C++

Purpose

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

Syntax



Notes:

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

where severity levels in order of increasing severity are:

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

See also “#pragma options” on page 325.

Notes

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

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

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

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

Example

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

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

Related References

“Compiler Command Line Options” on page 61

“flag” on page 130

“maxerr” on page 206

“#pragma options” on page 325

haltonmsg

C++

Purpose

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

Syntax

►► — -qhaltonmsg—=*msg_number*—————►►

Notes

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

Related References

“Compiler Command Line Options” on page 61

“Compiler Messages” on page 379

heapdebug

► C ► C++

Purpose

Enables debug versions of memory management functions.

Syntax

► — -q — noheapdebug — heapdebug — ►

Notes

The **-qheapdebug** options specifies that the debug versions of memory management functions (**_debug_malloc**, **_debug_malloc**, **new**, etc.) be used in place of regular memory management functions. This option defines the **__DEBUG_ALLOC__** macro.

By default, the compiler uses the regular memory management functions (**calloc**, **malloc**, **new**, etc.) and does not preinitialize their local storage.

This option makes the compiler search both `usr/vacpp/include` and `usr/include`.

Example

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

```
x1C -qheapdebug myprogram.c -o testing
```

Related References

“Compiler Command Line Options” on page 61

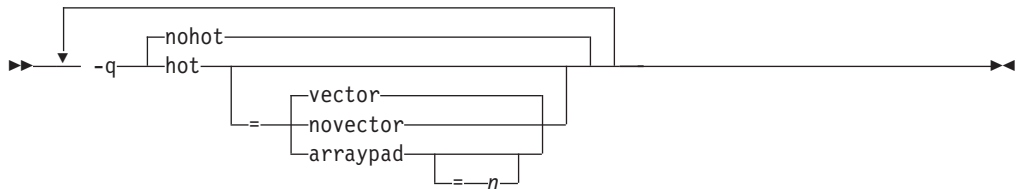
hot

C C++

Purpose

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

Syntax



where:

- arraypad The compiler will pad any arrays where it infers there may be a benefit and will pad by whatever amount it chooses. Not all arrays will necessarily be padded, and different arrays may be padded by different amounts.
- arraypad=*n* The compiler will pad every array in the code. The pad amount must be a positive integer value, and each array will be padded by an integral number of elements. Because *n* is an integral value, we recommend that pad values be multiples of the largest array element size, typically 4, 8, or 16.
- vector | novector The compiler converts certain operations that are performed in a loop on successive elements of an array (for example, square root, reciprocal square root) into a call to a library routine. This call will calculate several results at one time, which is faster than calculating each result sequentially.

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

Default

The **-qhot=vector** suboption is on by default when you specify the **-qhot**, **-qsmp**, **-O4**, or **-O5** options. If you do not specify at least level 2 of **-O** for **-qhot**, the compiler assumes **-O2**.

Notes

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

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

Example

The following example turns on the `-qhot=vector` option:

```
x1C -qhot=vector myprogram.c
```

Related References

“Compiler Command Line Options” on page 61

“C” on page 93

“O, optimize” on page 215

“smp” on page 252

hsflt

> C > C++

Purpose

Speeds up calculations by removing range checking on single-precision **float** results, and on conversions from floating point to integer. **-qnohsflt** specifies that single-precision expressions are rounded after expression evaluation, and that floating-point-to-integer conversions are to be checked for out of range values.

Syntax

► -q nohsflt hsflt ►

See also “#pragma options” on page 325.

Notes

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

The **-qhsflt** option overrides the **-qrndsngl** and **-qspnans** options.

The **-qhsflt** option is intended for specific applications in which floating-point computations have known characteristics. Using this option when compiling other application programs can produce incorrect results without warning.

Related References

- “Compiler Command Line Options” on page 61
- “float” on page 131
- “rndsngl” on page 243
- “spnans” on page 256
- “#pragma options” on page 325

hssngl

► C ► C++

Purpose

Specifies that single-precision expressions are rounded only when the results are stored into **float** memory locations. **-qnohssngl** specifies that single-precision expressions are rounded after expression evaluation. Using **-qhssngl** can improve run-time performance.

Syntax

► — -q —  —►

See also “#pragma options” on page 325.

Notes

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

Related References

“Compiler Command Line Options” on page 61

“float” on page 131

“#pragma options” on page 325

> C > C++

Purpose

Specifies an additional search path if the file name in the **#include** directive is not specified using its absolute path name.

Syntax

►— **-I**—*directory*—►

Notes

The value for *directory* must be a valid path name (for example, */u/golnaz*, or */tmp*, or *./subdir*). The compiler appends a slash (*/*) to the directory and then concatenates it with the file name before doing the search. The path *directory* is the one that the compiler searches first for **#include** files whose names do not start with a slash (*/*). If *directory* is not specified, the default is to search the standard directories.

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

The **-I** *directory* option can be specified more than once on the command line. If you specify more than one **-I** option, directories are searched in the order that they appear on the command line. See *Directory Search Sequence for Include Files Using Relative Path Names* for more information about searching directories.

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

Example

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

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

Related Tasks

“Compiler Command Line Options” on page 61

Related References

“Directory Search Sequence for Include Files Using Relative Path Names” on page 33

idirfirst

► C ► C++

Syntax

►► -q noidirfirst
idirfirst ◀◀

See also “#pragma options” on page 325.

Purpose

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

Notes

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

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

1. Search the directory where the current source file resides.
2. Search the directory or directories specified with the **-I** directory option.
3. Search the standard include directories, which are:
 - for C programs, **/usr/include**
 - for C++ programs, **/usr/vacpp/include** and **/usr/include**

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

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

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

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

The last valid **#pragma option** [NO]IDIRFirst remains in effect until replaced by a subsequent **#pragma option** [NO]IDIRFirst.

Example

To compile myprogram.c and search **/usr/tmp/myinclude** for included files before searching the current directory (where the source file resides), enter:

```
xlc myprogram.c -I/usr/tmp/myinclude -qidirfirst
```

Related References

“Compiler Command Line Options” on page 61

“I” on page 150

“stdinc” on page 260

“#pragma options” on page 325

ignerrno

> C > C++

Purpose

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

Syntax

►► — -q — noignerrno — ignerrno — ►►

See also “#pragma options” on page 325.

Notes

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

Related References

“Compiler Command Line Options” on page 61

“#pragma options” on page 325

ignprag

C C++

Purpose

Instructs the compiler to ignore certain pragma statements.

Syntax



where pragma statements affected by this option are:

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

See also “#pragma options” on page 325.

Notes

Suboptions are:

This option is useful for detecting aliasing pragma errors. Incorrect aliasing gives runtime errors that are hard to diagnose. When a runtime error occurs, but the error disappears when you use **-qignprag** with the **-O** option, the information specified in the aliasing pragmas is likely incorrect.

Example

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

```
x1C myprogram.c -qignprag=isolated
```

Related References

“Compiler Command Line Options” on page 61

“#pragma disjoint” on page 304

“#pragma isolated_call” on page 315

“#pragma options” on page 325

“Pragmas to Control Parallel Processing” on page 344

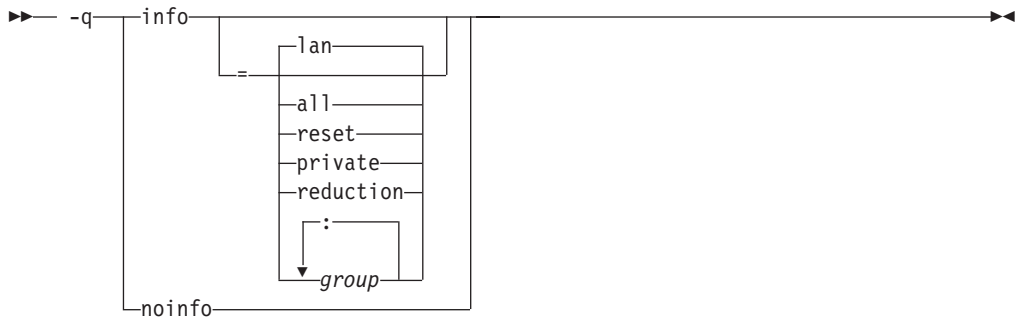
info

> C > C++

Purpose

Produces informational messages.

Syntax



where **-qinfo** options and diagnostic message groups are described in the **Notes** section below. See also “#pragma info” on page 311 and “#pragma options” on page 325.

Notes

Specifying **-qinfo** or **-qinfo=all** turns on all diagnostic messages for all groups except for the ppt (preprocessor trace) group in C++ code. Specifying **-qnoinfo** or **-qinfo=noall** turns off all diagnostic messages for all groups

Specifying **-qnoinfo** turns off all diagnostic messages.

You can use the **#pragma options info=suboption[:suboption ...]** or **#pragma options noinfo** forms of this compiler option to temporarily enable or disable messages in one or more specific sections of program code, and **#pragma options info=reset** to return to your initial **-qinfo** settings.

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

all Turns on all diagnostic messages for all groups.

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

> C++ The **-qinfo=all** option does not include the **ppt** group (preprocessor trace).

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

private Lists shared variables made private to a parallel loop.

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

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

<i>group</i>	Type of messages returned or suppressed
c99 noc99	C code that may behave differently between C89 and C99 language levels.
cls nocls	Classes
cmp nocmp	Possible redundancies in unsigned comparisons
cnd nocnd	Possible redundancies or problems in conditional expressions
cns nocns	Operations involving constants
cnv nocnv	Conversions
dcl nodcl	Consistency of declarations
eff noeff	Statements and pragmas with no effect
enu noenu	Consistency of enum variables
ext noext	Unused external definitions
gen nogen	General diagnostic messages
gnr nognr	Generation of temporary variables
got nogot	Use of goto statements
ini noini	Possible problems with initialization
inl noinl	Functions not inlined
lan nolan	Language level effects
obs noobs	Obsolete features
ord noord	Unspecified order of evaluation
par nopar	Unused parameters
por nopor	Nonportable language constructs
ppc noppc	Possible problems with using the preprocessor
ppt noppt	Trace of preprocessor actions
pro nopro	Missing function prototypes
rea norea	Code that cannot be reached
ret noret	Consistency of return statements
trd notrd	Possible truncation or loss of data or precision
tru notru	Variable names truncated by the compiler
trx notrx	Hexadecimal floating point constants rounding
uni nouni	Uninitialized variables
use nouse	Unused auto and static variables
vft novft	Generation of virtual function tables

Example

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

```
xlc myprogram.c -qinfo=all -qinfo=nocnv:norea
```

Related Concepts

"Reduction Operations in Parallelized Loops" on page 12

"Shared and Private Variables in a Parallel Environment" on page 13

Related References

"Compiler Command Line Options" on page 61

"#pragma info" on page 311

"#pragma options" on page 325

initauto

► C ► C++

Purpose

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

Syntax

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

See also “#pragma options” on page 325.

Notes

The option generates extra code to initialize the automatic (stack-allocated) storage of functions. It reduces the runtime performance of the program and should only be used for debugging.

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

Example

To compile `myprogram.c` so that automatic stack storage is initialized to hex value FF (decimal 255), enter:

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

Related References

“Compiler Command Line Options” on page 61

inlglue

> C > C++

Purpose

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

Syntax

► — -q — 

See also “#pragma options” on page 325.

Notes

Glue code, generated by the linker, is used for passing control between two external functions, or when you call functions through a pointer. It is also used to implement C++ virtual function calls. Therefore the **-qinlglue** option only affects function calls through pointers or calls to an external compilation unit. For calls to an external function, you should specify that the function is imported by using, for example, the **-qprocimported** option.

The inlining of glue code can cause the size of code to grow. This can be overridden by specifying the **-qcompact** option, thereby disabling the **-qinlglue** option.

Related References

“Compiler Command Line Options” on page 61

“compact” on page 101

“proclocal, procimported, procunknown” on page 233

“#pragma options” on page 325

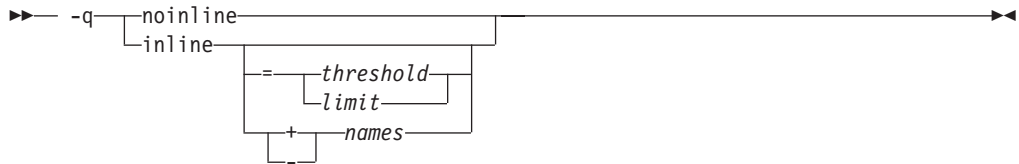
inline

C C++

Purpose

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

Syntax



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

-qinline The compiler attempts to inline all appropriate functions with 20 executable source statements or fewer, subject to any other settings of the suboptions to the **-qinline** option. If **-qinline** is specified last, all functions are inlined.

-qinline=threshold Sets a size limit on the functions to be inlined. The number of executable statements must be less than or equal to *threshold* for the function to be inlined. *threshold* must be a positive integer. The default value is 20. Specifying a threshold value of 0 causes no functions to be inlined except those functions marked with the **__inline**, **_Inline**, or **_inline** keywords.

The *threshold* value applies to logical C statements. Declarations are not counted, as you can see in the example below:

```
increment()
{
    int a, b, i;
    for (i=0; i<10; i++) /* statement 1 */
    {
        a=i;           /* statement 2 */
        b=i;           /* statement 3 */
    }
}
```

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

For example:

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

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

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

`-qinline+names` Attempts to inline the functions listed by *names* and any other appropriate functions. Each *name* must be separated by a colon (:). The option implies `-qinline`.

For example,

```
-qinline+food:clothes:vacation
```

causes all functions named **food**, **clothes**, or **vacation** to be inlined if possible, along with any other functions eligible for inlining.

A warning message is issued for functions that are not defined in the source file or that are defined but cannot be inlined.

This suboption overrides any setting of the *threshold* value. You can use a threshold value of zero along with `-qinline+names` to inline specific functions. For example:

```
-qinline=0
```

followed by:

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

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

`-qinline=limit` Specifies the maximum size (in bytes of generated code) to which a function can grow due to inlining. This limit does not affect the inlining of user specified functions.

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

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

<code>-qinline</code>	Compiler inlines all functions that it can.
<code>-qnoinline</code>	Compiler does not inline any functions.

Default

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

- If you specify the `-g` option (to generate debug information), no functions are inlined.
- If you optimize your program `-O`, the compiler attempts to inline all functions declared as inline. Otherwise, the compiler attempts to inline some of the simpler functions declared as inline.

Notes

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

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

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

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

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

The inline, _Inline, _inline, and __inline Function Specifiers: The C compiler provides keywords that you can use to specify functions that you want the compiler to inline:

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

For example:

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

causes `catherine` to be inlined, meaning that code is generated for the function, rather than a function call. The inline keywords also implicitly declare the function as static.

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

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

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

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

An inline function can be declared and defined simultaneously. If it is declared with one of the inline specifier keywords, it can be declared without a definition. The following code fragments shows an inline function definition. Note that the definition includes both the declaration and body of the inline function.

```
_inline int add(int i, int j) { return i + j; }  
  
inline double fahr(double t)
```

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

Example

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

```
x1C myprogram.c -O -qnoinline
```

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

```
x1C myprogram.c -O -qinline=12
```

Related References

"Compiler Command Line Options" on page 61

"O, optimize" on page 215

"Q" on page 236

ipa

► C ► C++

Purpose

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

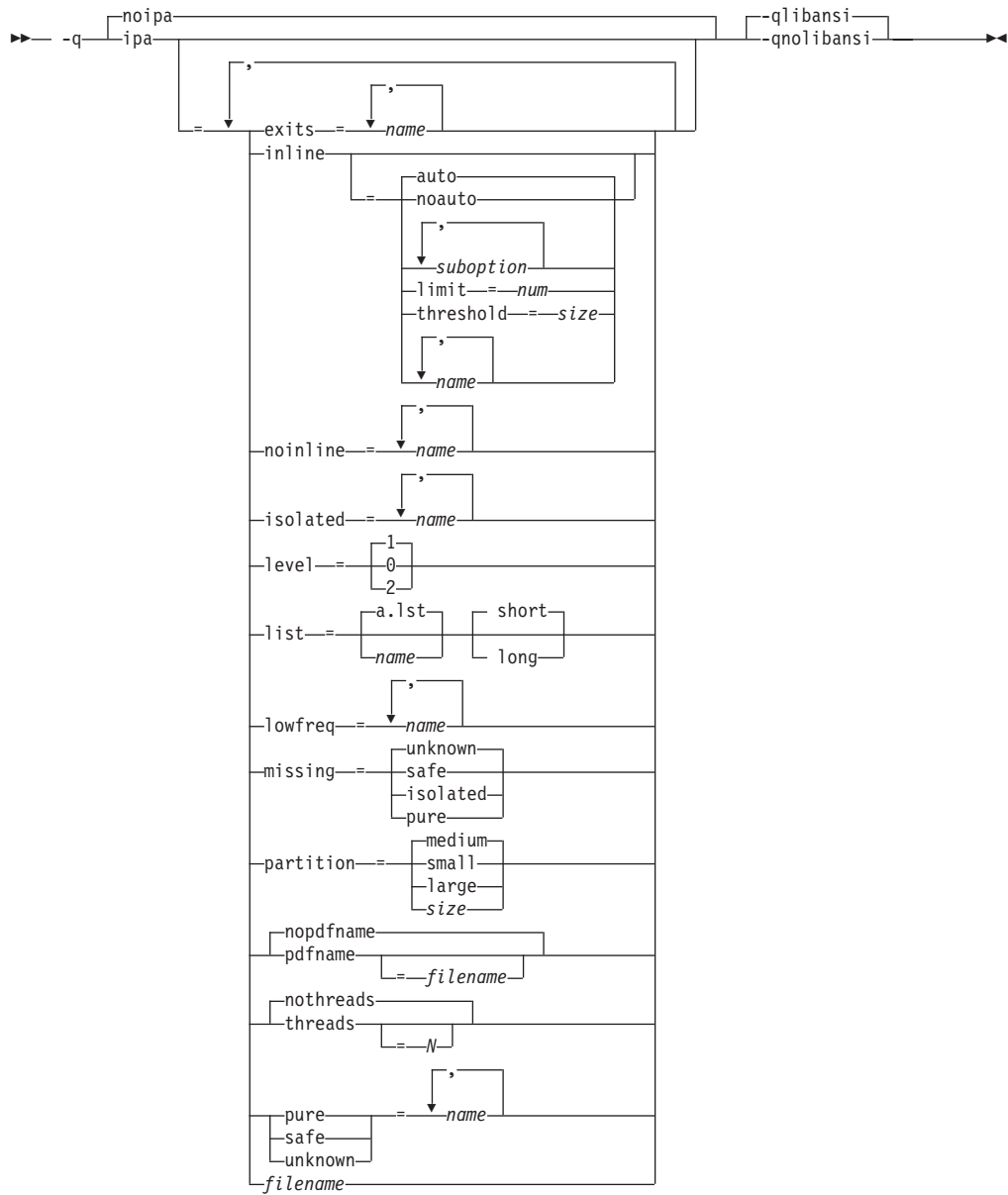
Compile-time syntax



where:

-qipa Compile-time Options	Description
-qipa	Activates interprocedural analysis with the following -qipa <i>suboption</i> defaults: <ul style="list-style-type: none">• inline=auto• level=1• missing=unknown• noprof• partition=medium
-qipa=object -qipa=noobject	Specifies whether to include standard object code in the object files. Specifying the noobject suboption can substantially reduce overall compile time by not generating object code during the first IPA phase. If the -S compiler option is specified with noobject , noobject is ignored. If compilation and linking are performed in the same step, and neither the -S nor any listing option is specified, -qipa=noobject is implied by default. If any object file used in linking with -qipa was created with the -qipa=noobject option, any file containing an entry point (the main program for an executable program, or an exported function for a library) must be compiled with -qipa .

Link-time syntax



where:

-qipa Link-time Options	Description
-qnoipa	Deactivates interprocedural analysis.
-qipa	Activates interprocedural analysis with the following -qipa <i>suboption</i> defaults: <ul style="list-style-type: none"> • inline=auto • level=1 • missing=unknown • noprof • partition=medium

-qipa Link-time Options	Description
-qlibansi -qnolibansi	The -qlibansi option assumes that all functions with the name of an ANSI C or C++ defined library function are in fact library functions. This is the default setting. The -qnolibansi option does not make this assumption.

Suboptions can also include one or more of the forms shown below. Separate multiple suboptions with commas.

Link-time Suboptions	Description
exits= <i>name</i> {, <i>name</i> }	Specifies names of functions which represent program exits. Program exits are calls which can never return and can never call any procedure which has been compiled with IPA pass 1.
inline=auto inline=noauto	Enables or disables automatic inlining only. The compiler still accepts user-specified functions as candidates for inlining.
inline[= <i>suboption</i>]	Same as specifying the -qinline compiler option, with <i>suboption</i> being any valid -qinline suboption.
inline=limit= <i>num</i>	Changes the size limits that the -Q option uses to determine how much inline expansion to do. This established limit is the size below which the calling procedure must remain. <i>number</i> is the optimizer's approximation of the number of bytes of code that will be generated. Larger values for this number allow the compiler to inline larger subprograms, more subprogram calls, or both. This argument is implemented only when inline=auto is on.
inline=threshold= <i>size</i>	Specifies the upper size limit of functions to be inlined, where <i>size</i> is a value as defined under inline=limit . This argument is implemented only when inline=auto is on.
inline= <i>name</i> {, <i>name</i> }	Specifies a comma-separated list of functions to try to inline, where functions are identified by <i>name</i> .
noinline= <i>name</i> {, <i>name</i> }	Specifies a comma-separated list of functions that must not be inlined, where functions are identified by <i>name</i> .
isolated= <i>name</i> {, <i>name</i> }	Specifies a list of <i>isolated</i> functions that are not compiled with IPA. Neither isolated functions nor functions within their call chain can refer to global variables.
level=0 level=1 level=2	Specifies the optimization level for interprocedural analysis. The default level is 1. Valid levels are as follows: <ul style="list-style-type: none"> • Level 0 - Does only minimal interprocedural analysis and optimization. • Level 1 - Turns on inlining, limited alias analysis, and limited call-site tailoring. • Level 2 - Performs full interprocedural data flow and alias analysis.

Link-time Suboptions	Description
<p>list</p> <p>list=[<i>name</i>]</p> <p>[short long]</p>	<p>Specifies that a listing file be generated during the link phase. The listing file contains information about transformations and analyses performed by IPA, as well as an optional object listing generated by the back end for each partition. This option can also be used to specify the name of the listing file.</p> <p>If listings have been requested (using either the -qlist or -qipa=list options), and <i>name</i> is not specified, the listing file name defaults to a.lst.</p> <p>The long and short suboptions can be used to request more or less information in the listing file. The short suboption, which is the default, generates the Object File Map, Source File Map and Global Symbols Map sections of the listing. The long suboption causes the generation of all of the sections generated through the short suboption, as well as the Object Resolution Warnings, Object Reference Map, Inliner Report and Partition Map sections.</p>
<p>lowfreq=<i>name</i>{,<i>name</i>}</p>	<p>Specifies names of functions which are likely to be called infrequently. These will typically be error handling, trace, or initialization functions. The compiler may be able to make other parts of the program run faster by doing less optimization for calls to these functions.</p>
<p>missing=<i>attribute</i></p>	<p>Specifies the interprocedural behavior of procedures that are not compiled with -qipa and are not explicitly named in an unknown, safe, isolated, or pure suboption.</p> <p>The following attributes may be used to refine this information:</p> <ul style="list-style-type: none"> • safe - Functions which do not indirectly call a visible (not missing) function either through direct call or through a function pointer. • isolated - Functions which do not directly reference global variables accessible to visible functions. Functions bound from shared libraries are assumed to be <i>isolated</i>. • pure - Functions which are <i>safe</i> and <i>isolated</i> and which do not indirectly alter storage accessible to visible functions. <i>pure</i> functions also have no observable internal state. • unknown - The default setting. This option greatly restricts the amount of interprocedural optimization for calls to <i>unknown</i> functions. Specifies that the missing functions are not known to be <i>safe</i>, <i>isolated</i>, or <i>pure</i>.
<p>partition=small</p> <p>partition=medium</p> <p>partition=large</p> <p>partition=<i>size</i></p>	<p>Specifies the size of each program partition created by IPA during pass 2.</p> <p>The size of the partition is directly proportional to the time required to link and the quality of the generated code. When partition sizes are large, the time to complete linkage is longer but the quality of the generated code is generally better. An integer may be used to specify partition <i>size</i> for finer control. This integer is in terms of unspecified units and its meaning may change from release to release. Its use should be limited to very short term tuning efforts.</p>

Link-time Suboptions	Description
pdfname pdfname= <i>filename</i>	<p>Specifies the name of the profile data file containing the PDF profiling information. If you do not specify <i>filename</i>, the default file name is <code>__pdf</code>.</p> <p>The profile is placed in the current working directory or in the directory named by the PDFDIR environment variable. This lets you do simultaneous runs of multiple executables using the same PDFDIR, which can be useful when tuning with PDF on dynamic libraries.</p>
nothreads threads threads= <i>N</i>	<p>Specifies the number of threads the compiler assigns to code generation.</p> <p>Specifying nothreads is equivalent to running one serial process. This is the default.</p> <p>Specifying threads allows the compiler to determine how many threads to use, depending on the number of processors available.</p> <p>Specifying threads=<i>N</i> instructs the program to use <i>N</i> threads. Though <i>N</i> can be any integer value in the range of 1 to MAXINT, <i>N</i> is effectively limited to the number of processors available on your system.</p>
pure= <i>name</i> {, <i>name</i> }	<p>Specifies a list of <i>pure</i> functions that are not compiled with -qipa. Any function specified as <i>pure</i> must be <i>isolated</i> and <i>safe</i>, and must not alter the internal state nor have side-effects, defined as potentially altering any data visible to the caller.</p>
safe= <i>name</i> {, <i>name</i> }	<p>Specifies a list of <i>safe</i> functions that are not compiled with -qipa. Safe functions can modify global variables, but may not call functions compiled with -qipa.</p>
unknown= <i>name</i> {, <i>name</i> }	<p>Specifies a list of <i>unknown</i> functions that are not compiled with -qipa. Any function specified as <i>unknown</i> can make calls to other parts of the program compiled with -qipa, and modify global variables and dummy arguments.</p>

Link-time Suboptions	Description
<i>filename</i>	<p>Gives the name of a file which contains suboption information in a special format.</p> <p>The file format is the following:</p> <pre data-bbox="821 344 1458 764"> # ... comment attribute{, attribute} = name{, name} missing = attribute{, attribute} exits = name{, name} lowfreq = name{, name} inline [= auto = noauto] inline = name{, name} [from name{, name}] inline-threshold = unsigned_integer inline-limit = unsigned_integer list [= file-name short long] noinline noinline = name{, name} [from name{, name}] level = 0 1 2 prof [= file-name] noprof partition = small medium large unsigned_integer </pre> <p>where <i>attribute</i> is one of:</p> <ul data-bbox="756 806 889 1008" style="list-style-type: none"> • exits • lowfreq • unknown • safe • isolated • pure

Notes

This option turns on or customizes a class of optimizations known as interprocedural analysis (IPA).

- IPA can significantly increase compilation time, even with the **-qipa=noobject** option, so using IPA should be limited to the final performance tuning stage of development.
- Specify the **-qipa** option on both the compile and link steps of the entire application, or as much of it as possible. You should at least compile the file containing **main**, or at least one of the entry points if compiling a library.
- While IPA's interprocedural optimizations can significantly improve performance of a program, they can also cause previously incorrect but functioning programs to fail. Listed below are some programming practices that can work by accident without aggressive optimization, but are exposed with IPA:
 1. Relying on the allocation order or location of automatics. For example, taking the address of an automatic variable and then later comparing it with the address of another local to determine the growth direction of a stack. The C language does not guarantee where an automatic variable is allocated, or its position relative to other automatics. Do not compile such a function with IPA (and expect it to work).
 2. Accessing an either invalid pointer or beyond an array's bounds. IPA can reorganize global data structures. A wayward pointer which may have previously modified unused memory may now trample upon user allocated storage.
- Ensure you have sufficient resources to compile with IPA. IPA can generate significantly larger object files than traditional compilers. As a result, the temporary storage used to hold these intermediate files (by convention /tmp on

AIX) is sometimes too small. If a large application is being compiled, consider redirecting temporary storage with the TMPDIR environment variable.

- Ensure there is enough swap space to run IPA (at least 200Mb for large programs). Otherwise the operating system might kill IPA with a signal 9, which cannot be trapped, and IPA will be unable to clean up its temporary files.
- You can link objects created with different releases of the compiler, but you must ensure that you use a linker that is at least at the same release level as the newer of the compilers used to create the objects being linked.
- Some symbols which are clearly referenced or set in the source code may be optimized away by IPA, and may be lost to debug, nm, or dump outputs. Using IPA together with the **-g** compiler will usually result in non-steppable output.

The necessary steps to use IPA are:

1. Do preliminary performance analysis and tuning before compiling with the **-qipa** option, because the IPA analysis uses a two-pass mechanism that increases compile and link time. You can reduce some compile and link overhead by using the **-qipa=noobject** option.
2. Specify the **-qipa** option on both the compile and the link steps of the entire application, or as much of it as possible. Use suboptions to indicate assumptions to be made about parts of the program *not* compiled with **-qipa**. During compilation, the compiler stores interprocedural analysis information in the **.o** file. During linking, the **-qipa** option causes a complete recompilation of the entire application.

Note: If a Severe error occurs during compilation, **-qipa** returns RC=1 and terminates. Performance analysis also terminates.

Example

To compile a set of files with interprocedural analysis, enter:

```
x1C -c -O3 *.c -qipa
x1C -o product *.o -qipa
```

Here is how you might compile the same set of files, improving the optimization of the second compilation, and the speed of the first compile step. Assume that there exists two functions, *trace_error* and *debug_dump*, which are rarely executed.

```
x1C -c -O3 *.c -qipa=noobject
x1C -c - *.o -qipa=lowfreq=trace_error,debug_dump
```

Related References

“Compiler Command Line Options” on page 61

“libansi” on page 191

“list” on page 193

“pdf1, pdf2” on page 226

“S” on page 248

isolated_call

C C++

Purpose

Specifies functions in the source file that have no side effects.

Syntax

```
►► -q-isolated_call=function_name◄◄
```

where:

function_name Is the name of a function that does not have side effects, except changing the value of a variable pointed to by a pointer or reference parameter, or does not rely on functions or processes that have side effects.

Side effects are any changes in the state of the runtime environment. Examples of such changes are accessing a volatile object, modifying an external object, modifying a file, or calling another function that does any of these things. Functions with no side effects cause no changes to external and static variables.

function_name can be a list of functions separated by colons (:).

See also “#pragma isolated_call” on page 315 and “#pragma options” on page 325.

Notes

Marking a function as isolated can improve the runtime performance of optimized code by indicating the following to the optimizer:

- external and static variables are not changed by the called function
- calls to the function with loop-invariant parameters may be moved out of loops
- multiple calls to the function with the same parameter may be merged into one call
- calls to the function may be discarded if the result value is not needed

The **#pragma options** keyword **isolated_call** must be specified at the top of the file, before the first C or C++ statement. You can use the **#pragma isolated_call** directive at any point in your source file.

Example

To compile myprogram.c, specifying that the functions myfunction(int) and classfunction(double) do not have side effects, enter:

```
x1C myprogram.c -qisolated_call=myfunction:classfunction
```

Related References

“Compiler Command Line Options” on page 61

“#pragma isolated_call” on page 315

“#pragma options” on page 325

keepinlines

C++

Purpose

Instructs the compiler to keep or discard definitions for unreferenced extern inline functions.

Syntax

►► — -q —  —►►

Notes

The default **-qnokeepinlines** setting instructs the compiler to discard the definitions of unreferenced extern inline functions. This can reduce the size of the object files.

The **-qkeepinlines** setting keeps the definitions of unreferenced extern inline functions. This setting provides the same behavior as VisualAge C++ compilers previous to the v5.0.2.1 update level, allowing compatibility with shared libraries and object files built with the earlier releases of the compiler.

Related References

“Compiler Command Line Options” on page 61

“inline” on page 159

“The inline, _Inline, _inline, and __inline Function Specifiers” on page 161

keyword

> C > C++

Purpose

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

Syntax

►► -q keyword
nokeyword == *keyword_name* ◀◀

Notes

By default all the built-in keywords defined in the C and C++ language standards are reserved as keywords. You cannot add keywords to the language with this option. However, you can use **-qnokeyword=keyword_name** to disable built-in keywords, and use **-qkeyword=keyword_name** to reinstate those keywords.

This option can be used with all C++ built-in keywords.

This option can also be used with the following C built-in keywords:

- asm
- restrict
- typeof

Example

You can reinstate bool with the following invocation:

```
xlc -qkeyword=bool
```

Related References

“Compiler Command Line Options” on page 61

L

► C ► C++

Purpose

Searches the path directory for library files specified by the *-lkey* option.

Syntax

►► — *-L—directory* —————►►

Notes

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

Default

The default is to search only the standard directories.

Example

To compile `myprogram.c` so that the directory `/usr/tmp/old` and all other directories specified by the *-l* option are searched for the library `libspfiles.a`, enter:

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

Related References

“Compiler Command Line Options” on page 61

“l” on page 174

A set of two dark rectangular tabs with white text. The first tab is labeled 'C' and the second tab is labeled 'C++'. The 'C++' tab is currently selected and highlighted.

Purpose

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

Syntax

►► — *-l*—*key*—————▶▶

Default

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

Notes

The actual search path can be modified with the *-Ldirectory* or *-Z* options. See *-B*, *-brtl*, and *-bstatic,-bdynamic* for information on specifying the types of libraries that are searched (for static or dynamic linking).

Example

To compile *myprogram.c* and include my library (*libmylibrary.a*), enter:

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

Related Tasks

“Specify Compiler Options in a Configuration File” on page 27

Related References

“Compiler Command Line Options” on page 61

“B” on page 88

“b” on page 89

“brtl” on page 92

“l”

“Z” on page 296

langlvl

► C ► C++

Purpose

Selects the language level for the compilation.

Syntax

►► -q—langlvl—=*language*◀◀

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

See also “#pragma langlvl” on page 317 and “#pragma options” on page 325.

Default

The default language level is **ansi** when using **xlc** or **c89** to invoke the compiler, and **extended** when using **x1C** or **cc**. The **extended** language level is based on C89.

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

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

The **pragma** directive must appear before any noncommentary lines in the source code.

Notes

► C For C programs, you can use the following **-qlanglvl** suboptions for *language*:

ansi	Compilation conforms to the ANSI C89 standard.
classic	Allows the compilation of non-ANSI programs, and conforms closely to the K&R level preprocessor. This language level is not supported by the AIX v5.1 system header files, such as math.h. If you must use the AIX v5.1 system header files, consider compiling your program to the ansi or extended language levels.
extended	Provides compatibility with the RT compiler and classic . This language level is based on C89.
saa12	Compilation conforms to the SAA C Level 2 CPI language definition, with some exceptions.
saa	Compilation conforms to the current SAA C CPI language definition. This is currently SAA C Level 2.
stdc89	Compilation conforms to the ANSI C89 standard.
stdc99	Compilation conforms to the ISO C99 standard. Note: Not all operating system releases support the header files and runtime library required by C99.
extc89	Compilation conforms to the ANSI C89 standard, and accepts implementation-specific language extensions.
extc99	Compilation conforms to the ISO C99 standard, and accepts implementation-specific language extensions. Note: Not all operating system releases support the header files and runtime library required by C99.

[no]ucs Under language levels **stdc99** and **extc99**, the default is **-qlanglvl=ucs**

This option controls whether Unicode characters are allowed in identifiers, string literals and character literals in program source code.

The Unicode character set is supported by the C standard. This character set contains the full set of letters, digits and other characters used by a wide range of languages, including all North American and Western European languages. Unicode characters can be 16 or 32 bits. The ASCII one-byte characters are a subset of the Unicode character set.

When this option is set to yes, you can insert Unicode characters in your source files either directly or using a notation that is similar to escape sequences. Because many Unicode characters cannot be displayed on the screen or entered from the keyboard, the latter approach is usually preferred. Notation forms for Unicode characters are `\uhhhh` for 16-bit characters, or `\Uhhhhhhhhh` for 32-bit characters, where *h* represents a hexadecimal digit. Short identifiers of characters are specified by ISO/IEC 10646.

The following **-qlanglvl** suboptions are accepted but ignored by the C compiler. Use **-qlanglvl=extended**, **-qlanglvl=extc99**, or **-qlanglvl=extc89** to enable the functions that these suboptions imply. For other values of **-qlanglvl**, the functions implied by these suboptions are disabled.

[no]gnu_assert	GNU C portability option.
[no]gnu_explicitregvar	GNU C portability option.
[no]gnu_include_next	GNU C portability option.
[no]gnu_locallabel	GNU C portability option.
[no]gnu_warning	GNU C portability option.

C++ For C++ programs, you can specify one or more of the following **-qlanglvl** suboptions for *language*:

ansi	Compilation conforms to the C89 standard for C programs, and the ANSI C++ standard for C++ programs.
compat366	Compilation conforms to the IBM C and C++ Compilers V 3.6.
extended	Compilation is the same as ansi mode, with some differences to accommodate extended language features.
strict98	Compilation conforms to the ANSI C standard for C programs, and the ANSI C++ standard for C++ programs.

[no]anonstruct

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

By default, VisualAge C++ allows anonymous structs. This is an extension to the C++ standard and gives behavior that is compatible with the C++ compilers provided by Microsoft Visual C++.

Anonymous structs typically are used in unions, as in the following code fragment:

```
union U {
    struct {
        int i:16;
        int j:16;
    };
    int k;
} u;
// ...
u.j=3;
```

When this suboption is set, you receive a warning if your code declares an anonymous struct and **-qinfo=por** is specified. When you build with **-qlanglvl=noanonstruct**, an anonymous struct is flagged as an error. Specify **noanonstruct** for compliance with standard C++.

[no]ansifor

This suboption controls whether scope rules defined in the C++ standard apply to names declared in for-init statements.

By default, standard C++ rules are used. For example the following code causes a name lookup error:

```
{
    //...
    for (int i=1; i<5; i++) {
        cout << i * 2 << endl;
    }
    i = 10; // error
}
```

The reason for the error is that *i*, or any name declared within a for-init-statement, is visible only within the for statement. To correct the error, either declare *i* outside the loop or set `ansiForStatementScopes` to `no`.

Set **noansifor** to allow old language behavior. You may need to do this for code that was developed with other products, such as the compilers provided by earlier versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.

[no]ansisinit

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

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

The default setting is **ansisinit**.

[no]gnu_assert	GNU C portability option to enable or disable support for the following GNU C system identification assertions: <ul style="list-style-type: none"> • #assert • #unassert • #cpu • #machine • #system
[no]gnu_explicitregvar	GNU C portability option to control whether the compiler accepts and ignores the specification of explicit registers for variables.
[no]gnu_include_next	GNU C portability option to enable or disable support for the GNU C #include_next preprocessor directive.
[no]gnu_locallabel	GNU C portability option to enable or disable support for locally-declared labels.
[no]gnu_warning	GNU C portability option to enable or disable support for the GNU C #warning preprocessor directive.
[no]oldfriend	This option controls whether friend declarations that name classes without elaborated class names are treated as C++ errors.

By default, VisualAge C++ lets you declare a friend class without elaborating the name of the class with the keyword `class`. This is an extension to the C++ standard and gives behavior that is compatible with the C++ compilers provided by earlier versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.

For example, the statement below declares the class `IFont` to be a friend class and is valid when the **oldfriend** suboption is set specified.

```
friend IFont;
```

Set the **nooldfriend** suboption for compliance with standard C++. The example declaration above causes a warning unless you modify it to the statement as below, or suppress the warning message with **-qsuppress** option.

```
friend class IFont;
```

[no]oldmath	This suboption controls which versions of math function declarations in <code><math.h></code> are included when you specify <code>math.h</code> as an included or primary source file. <p>By default, the new standard math functions are used. Build with -qlanglvl=nooldmath for strict compliance with the C++ standard.</p> <p>For compatibility with modules that were built with earlier versions of VisualAge C++ and predecessor products you may need to build with -qlanglvl=oldmath.</p>
-------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

[no]oldtempacc

This suboption controls whether access to a copy constructor to create a temporary object is always checked, even if creation of the temporary object is avoided.

By default, VisualAge C++ suppresses the access checking. This is an extension to the C++ standard and gives behavior that is compatible with the C++ compilers provided by VisualAge C++ for OS/2 3.0, VisualAge for C++ for Windows, Version 3.5, and Microsoft Visual C++.

When this suboption is set to yes, you receive a warning if your code uses the extension, unless you disable the warning message with the **-qsuppress** option.

Set **-qlanglvl=nooldtempacc** for compliance with standard C++. For example, the throw statement in the following code causes an error because the copy constructor is a protected member of class C:

```
class C {
public:
    C(char *);
protected:
    C(const C&);
};

C foo() {return C("test");} // returns a copy of a C object
void f()
{
    // catch and throw both make implicit copies of the thrown object
    throw C("error"); // throws a copy of a C object
    const C& r = foo(); // uses the copy of a C object created by
}
```

The example code above contains three ill formed uses of the copy constructor C(const C&).

[no]oldtmplalign

This suboption specifies the alignment rules implemented in versions of the compiler (x1C) prior to Version 5.0. These earlier versions of the x1C compiler ignore alignment rules specified for nested templates. By default, these alignment rules are not ignored in VisualAge C++ 4.0 or later. For example, given the following template the size of A<char>::B will be 5 with **-qlanglvl=nooldtmplalign**, and 8 with **-qlanglvl=oldtmplalign** :

```
template <class T>
struct A {
#pragma options align=packed
    struct B {
        T m;
        int m2;
    };
#pragma options align=reset
};
```

[no]oldtmpspec

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

By default, VisualAge C++ allows these old specializations (**-qlanglvl=nooldtmpspec**). This is an extension to standard C++ and gives behavior that is compatible with the C++ compilers provided by VisualAge C++ for OS/2 3.0, VisualAge for C++ for Windows, Version 3.5, and Microsoft Visual C++.

When **-qlanglvl=oldtmpspec** is set, you receive a warning if your code uses the extension, unless you suppress the warning message with the **-qsuppress** option.

For example, you can explicitly specialize the template class `ribbon` for type `char` with the following lines:

```
template<class T> class ribbon { /*...*/};  
class ribbon<char> { /*...*/};
```

Set **-qlanglvl=nooldtmpspec** for compliance with standard C++. In the example above, the template specialization must be modified to:

```
template<class T> class ribbon { /*...*/};  
template<> class ribbon<char> { /*...*/};
```

[no]anonunion

This suboption controls what members are allowed in anonymous unions.

When this suboption is set to **anonunion**, anonymous unions can have members of all types that standard C++ allows in non-anonymous unions. For example, non-data members, such as structs, typedefs, and enumerations are allowed.

Member functions, virtual functions, or objects of classes that have non-trivial default constructors, copy constructors, or destructors cannot be members of a union, regardless of the setting of this option.

By default, VisualAge C++ allows non-data members in anonymous unions. This is an extension to standard C++ and gives behavior that is compatible with the C++ compilers provided by previous versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.

When this option is set to **anonunion**, you receive a warning if your code uses the extension, unless you suppress the warning message with the **-qsuppress** option.

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

[no]illptom

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

By default, VisualAge C++ allows these forms. This is an extension to standard C++ and gives behavior that is compatible with the C++ compilers provided by earlier versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.

When this suboption is set to **illptom**, you receive warnings if your code uses the extension, unless you suppress the warning messages with the **-qsuppress** option.

For example, the following code defines a pointer to a function member, `p`, and initializes it to the address of `C::foo`, in the old style:

```
struct C {  
    void foo(int);  
};  
  
void (C::*p) (int) = C::foo;
```

Set **noillptom** for compliance with the C++ standard. The example code above must be modified to use the `&` operator.

```
struct C {  
    void foo(int);  
};  
  
void (C::*p) (int) = &C::foo;
```

[no]implicitint

This suboption controls whether VisualAge C++ will accept missing or partially specified types as implicitly specifying int. This is no longer accepted in the standard but may exist in legacy code.

With the suboption set to **noimplicitint**, all types must be fully specified.

With the suboption set to **implicitint**, a function declaration at namespace scope or in a member list will implicitly be declared to return int. Also, any declaration specifier sequence that does not completely specify a type will implicitly specify an integer type. Note that the effect is as if the int specifier were present. This means that the specifier const, by itself, would specify a constant integer.

The following specifiers do not completely specify a type.

- auto
- const
- extern
- extern "<literal>"
- inline
- mutable
- friend
- register
- static
- typedef
- virtual
- volatile
- platform specific types (for example, _cdecl)

Note that any situation where a type is specified is affected by this suboption. This includes, for example, template and parameter types, exception specifications, types in expressions (eg, casts, dynamic_cast, new), and types for conversion functions.

By default, VisualAge C++ sets **-qlanglvl=implicitint**. This is an extension to the C++ standard and gives behavior that is compatible with the C++ compilers provided by earlier versions of VisualAge C++ and predecessor products, and Microsoft Visual C++.

For example, the return type of function MyFunction is int because it was omitted in the following code:

```
MyFunction()
{
    return 0;
}
```

Set **-qlanglvl=noimplicitint** for compliance with standard C++. For example, the function declaration above must be modified to:

```
int MyFunction()
{
    return 0;
}
```

[no]newexcp

This suboption determines whether or not the C++ operator new throws an exception. The standard exception `std::bad_alloc` can be thrown when the requested memory allocation fails. This option does not apply to the nothrow versions of the new operator.

The standard implementation of the new operators fully support exceptions. For compatibility with previous versions of VisualAge C++, these operators return 0 by default.

[no]offsetnonpod

This suboption controls whether the `offsetof` macro can be applied to classes that are not data-only. C++ programmers often casually call data-only classes “Plain Old Data” (POD) classes.

By default, VisualAge C++ allows `offsetof` to be used with nonPOD classes. This is an extension to the C++ standard, and gives behavior that is compatible with the C++ compilers provided by VisualAge C++ for OS/2 3.0, VisualAge for C++ for Windows, Version 3.5, and Microsoft Visual C++

When this option is set, you receive a warning if your code uses the extension, unless you suppress the warning message with the `-qsuppress` option.

Set `-qlanglvl=nooffsetnonpod` for compliance with standard C++.

Set `-qlanglvl=offsetnonpod` if your code applies `offsetof` to a class that contains one of the following:

- user-declared constructors or destructors
- user-declared assignment operators
- private or protected non-static data members
- base classes
- virtual functions
- non-static data members of type pointer to member
- a struct or union that has non-data members
- references

[no]olddigraph

This option controls whether old-style digraphs are allowed in your C++ source. It applies only when `-qdigraph` is also set.

By default, VisualAge C++ supports only the digraphs specified in the C++ standard.

Set `-qlanglvl=olddigraph` if your code contains at least one of following digraphs:

Digraph

Resulting Character

%% # (pound sign)

%%%%

(double pound sign, used as the preprocessor macro concatenation operator)

Set `-qlanglvl=noolddigraph` for compatibility with standard C++ and the extended C++ language level supported by previous versions of VisualAge C++ and predecessor products.

<code>[no]traienum</code>	<p>This suboption controls whether trailing commas are allowed in enum declarations.</p> <p>By default, the compiler allows one or more trailing commas at the end of the enumerator list. This is an extension to the C++ standard, and provides compatibility with Microsoft Visual C++. The following enum declaration uses this extension:</p> <pre>enum grain { wheat, barley, rye,, };</pre> <p>Set -qlanglvl=notraienum for compliance with standard C++ or with the ANSI language level supported by previous versions of VisualAge C++ and predecessor products.</p>
<code>[no]typedefclass</code>	<p>This suboption provides backwards compatibility with previous versions of VisualAge C++ and predecessor products.</p> <p>The current C++ standard does not allow a typedef name to be specified where a class name is expected. This option relaxes that restriction. Set -qlanglvl=typedefclass to allow the use of typedef names in base specifiers and constructor initializer lists.</p> <p>By default, a typedef name cannot be specified where a class name is expected.</p>
<code>[no]ucs</code>	<p>This suboption controls whether Unicode characters are allowed in identifiers, string literals and character literals in C++ sources.</p> <p>The Unicode character set is supported by the C++ standard. This character set contains the full set of letters, digits and other characters used by a wide range of languages, including all North American and Western European languages. Unicode characters can be 16 or 32 bits. The ASCII one-byte characters are a subset of the Unicode character set.</p> <p>When -qlanglvl=ucs is set, you can insert Unicode characters in your source files either directly or using a notation that is similar to escape sequences. Because many Unicode characters cannot be displayed on the screen or entered from the keyboard, the latter approach is usually preferred. Notation forms for Unicode characters are <code>\uhhhh</code> for 16-bit characters, or <code>\Uhhhhhhhh</code> for 32-bit characters, where <i>h</i> represents a hexadecimal digit. Short identifiers of characters are specified by ISO/IEC 10646.</p> <p>Under language levels stdc99 and extc99, the default setting of -qlanglvl=ucs can be disabled by specifying -qlanglvl=noucs.</p>

[no]zeroextarray

This suboption controls whether zero-extent arrays are allowed as the last non-static data member in a class definition.

By default, the compiler allows arrays with zero elements. This is an extension to the C++ standard, and provides compatibility with Microsoft Visual C++. The example declarations below define dimensionless arrays a and b.

```
struct S1 { char a[0]; };  
struct S2 { char b[]; };
```

Set **nozeroextarray** for compliance with standard C++ or with the ANSI language level supported by previous versions of VisualAge C++ and predecessor products.

When this option is set, you receive warnings about zero-extent arrays in your code, unless you suppress the warning message with the **-qsuppress** option.

Exceptions to the **ansi** mode addressed by **classic** are as follows:

Tokenization Tokens introduced by macro expansion may be combined with adjacent tokens in some cases. Historically, this was an artifact of the text-based implementations of older preprocessors, and because, in older implementations, the preprocessor was a separate program whose output was passed on to the compiler.

For similar reasons, tokens separated only by a comment may also be combined to form a single token. Here is a summary of how tokenization of a program compiled in **classic** mode is performed:

1. At a given point in the source file, the next token is the longest sequence of characters that can possibly form a token. For example, `i++++j` is tokenized as `i ++ ++ + j` even though `i ++ + ++ j` may have resulted in a correct program.
2. If the token formed is an identifier and a macro name, the macro is replaced by the text of the tokens specified on its **#define** directive. Each parameter is replaced by the text of the corresponding argument. Comments are removed from both the arguments and the macro text.
3. Scanning is resumed at the first step from the point at which the macro was replaced, as if it were part of the original program.
4. When the entire program has been preprocessed, the result is scanned again by the compiler as in the first step. The second and third steps do not apply here since there will be no macros to replace. Constructs generated by the first three steps that resemble preprocessing directives are not processed as such.

It is in the third and fourth steps that the text of adjacent but previously separate tokens may be combined to form new tokens.

The `\` character for line continuation is accepted only in string and character literals and on preprocessing directives.

Constructs such as:

```
#if 0
  "unterminated
#endif
#define US "Unterminating string
char *s = US terminated now"
```

will not generate diagnostic messages, since the first is an unterminated literal in a **FALSE** block, and the second is completed after macro expansion. However:

```
char *s = US;
```

will generate a diagnostic message since the string literal in **US** is not completed before the end of the line.

Empty character literals are allowed. The value of the literal is zero.

Preprocessing directives

The # token must appear in the first column of the line. The token immediately following # is available for macro expansion. The line can be continued with \ only if the name of the directive and, in the following example, the (has been seen:

```
#define f(a,b) a+b
f\
(1,2)      /* accepted */
#define f(a,b) a+b
f\
1,2)      /* not accepted */
```

The rules concerning \ apply whether or not the directive is valid. For example,

```
#\
define M 1  /* not allowed */
#def\
ine M 1     /* not allowed */
#define\
M 1        /* allowed */
#dfine\
M 1        /* equivalent to #define M 1, even
            though #dfine is not valid */
```

Following are the preprocessor directive differences between **classic** mode and **ansi** mode. Directives not listed here behave similarly in both modes.

#ifdef/#ifndef

When the first token is not an identifier, no diagnostic message is generated, and the condition is FALSE.

#else When there are extra tokens, no diagnostic message is generated.

#endif When there are extra tokens, no diagnostic message is generated.

#include

The < and > are separate tokens. The header is formed by combining the spelling of the < and > with the tokens between them. Therefore /* and // are recognized as comments (and are always stripped), and the " and ' do begin literals within the < and >. (Remember that in C programs, C++-style comments // are recognized when **-qcplusplusmt** is specified.)

#line The spelling of all tokens which are not part of the line number form the new file name. These tokens need not be string literals.

#error Not recognized in **classic** mode.

#define

A valid macro parameter list consists of zero or more identifiers each separated by commas. The commas are ignored and the parameter list is constructed as if they were not specified. The parameter names need not be unique. If there is a conflict, the last name specified is recognized.

For an invalid parameter list, a warning is issued. If a macro name is redefined with a new definition, a warning will be issued and the new definition used.

#undef When there are extra tokens, no diagnostic message is generated.

Macro expansion

- When the number of arguments on a macro invocation does not match the number of parameters, a warning is issued.
- If the (token is present after the macro name of a function-like macro, it is treated as too few arguments (as above) and a warning is issued.
- Parameters are replaced in string literals and character literals.
- Examples:

```
#define M()    1
#define N(a)  (a)
#define O(a,b) ((a) + (b))

M(); /* no error */
N(); /* empty argument */
O(); /* empty first argument
      and too few arguments */
```

Text Output No text is generated to replace comments.

Related References

“Compiler Command Line Options” on page 61

“bitfields” on page 90

“chars” on page 97

“flag” on page 130

“inline” on page 159

“M” on page 198

“ro” on page 244

“suppress” on page 263

“#pragma langlvl” on page 317

“#pragma options” on page 325

largepage

► C ► C++

Purpose

Instructs the compiler to exploit large page heaps available on Power 4 systems running AIX v5.1D or later.

Syntax

►► -q no|largepage | largepage ◀◀

Notes

Compiling with **-qlargepage** can result in improved program performance. This option has effect only on Power 4 systems running AIX v5.1D or later.

This option is only valid when used together with IPA (**-qipa**, **-O4**, **-O5** compiler options).

Example

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

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

Related References

“Compiler Command Line Options” on page 61

“ipa” on page 163

ldbl128, longdouble

C C++

Purpose

Increases the size of **long double** type from 64 bits to 128 bits.

Syntax



See also “#pragma options” on page 325.

Notes

The **-q1ongdouble** option is the same as the **-qldbl128** option.

Separate libraries are provided that support 128-bit **long double** types. These libraries will be automatically linked if you use any of the invocation commands with the **128** suffix (**x1C128**, **xlc128**, **cc128**, **x1C128_r**, **xlc128_r**, or **cc128_r**). You can also manually link to the 128-bit versions of the libraries using the **-lkey** option, as shown in the following table:

Default (64-bit) long double		128-bit long double	
Library	Form of the <i>-lkey</i> option	Library	Form of the <i>-lkey</i> option
libC.a	-lC	libC128.a	-lC128
libC_r.a	-lC_r	libC128_r.a	-lC128_r

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

The **-qldbl128** option defines **__LONGDOUBLE128**.

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

Example

To compile myprogram.c so that **long double** types are 128 bits, enter:

```
x1C myprogram.c -qldbl128 -lC128
```

or:

```
x1C128 myprogram.c
```

Related References

“Compiler Command Line Options” on page 61

“1” on page 174

“#pragma options” on page 325

libansi

► C ► C++

Purpose

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

Syntax

►► -q no1ibansi 1ibansi ►►

See also “#pragma options” on page 325.

Notes

This will allow the optimizer to generate better code because it will know about the behavior of a given function, such as whether or not it has any side effects.

Related References

“Compiler Command Line Options” on page 61

“#pragma options” on page 325

linedebug

C C++

Purpose

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

Syntax

→ -q no|linedebug →

Notes

This option produces minimal debugging information, so the resulting object size is smaller than that produced if the **-g** debugging option is specified. You can use the debugger to step through the source code, but you will not be able to see or query variable information. The traceback table, if generated, will include line numbers.

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

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

The **-g** option overrides the **-qlinedebug** option. If you specify **-g -qnolinedebug** on the command line, **-qnolinedebug** is ignored and the following warning is issued:

```
1506-... (W) Option -qnolinedebug is incompatible with option -g and is ignored
```

Example

To compile myprogram.c to produce an executable program **testing** so you can step through it with a debugger, enter:

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

Related References

“Compiler Command Line Options” on page 61

“g” on page 141

“O, optimize” on page 215

“Q” on page 236

“#pragma options” on page 325

list

► C ► C++

Purpose

Produces a compiler listing that includes an object listing.

Syntax

► — -q —  — ►

See also “#pragma options” on page 325.

Notes

For C, options that are not defaults appear in all listings, even if **nolist** is specified. The **noprint** option overrides this option. This does not apply to C++.

Example

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

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

Related References

“Compiler Command Line Options” on page 61

“print” on page 231

“#pragma options” on page 325

listopt

> C > C++

Purpose

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

The listing will show options in effect as set by the compiler default, configuration file, and command line settings. Option settings caused by **#pragma** statements in the program source are not shown in the compiler listing.

Syntax

► — -q —  — ►

Example

To compile myprogram.c to produce a compiler listing that shows all options in effect, enter:

```
x1C myprogram.c -q listopt
```

Related References

“Compiler Command Line Options” on page 61

“Resolving Conflicting Compiler Options” on page 31

longlit

C C++

Purpose

Makes unaffixed literals into the long type in 64-bit mode.

Syntax

► -q no longlit / longlit ►

Notes

The following table shows the implicit types for constants in 64-bit mode when compiling in the **stdc89**, **extc89**, or **extended** language level:

	default 64-bit mode	64-bit mode with qlonglit
unaffixed decimal	signed int signed long unsigned long	signed long unsigned long
unaffixed octal or hex	signed int unsigned int signed long unsigned long	signed long unsigned long
affixed by u/U	unsigned int unsigned long	unsigned long
affixed by l/L	signed long unsigned long	signed long unsigned long
affixed by ul/UL	unsigned long	unsigned long

The following table shows the implicit types for constants in 64-bit mode when compiling in the **stdc99** or **extc99** language level:

	Decimal Constant	- qlonglit effect on Decimal Constant
unaffixed	int long int	long int
u or U	unsigned int unsigned long int	unsigned long int
l or L	long int	long int
Both u or U , and l or L	unsigned long int	unsigned long int
ll or LL	long long int	long long int
Both u or U , and ll or LL	unsigned long long int	unsigned long long int

	Octal or Hexadecimal Constant	-qlonglit effect on Octal or Hexadecimal Constant
unsuffixed	int unsigned int long int unsigned long int	long int unsigned long int
u or U	unsigned int unsigned long int	unsigned long int
l or L	long int unsigned long int	long int unsigned long int
Both u or U , and l or L	unsigned long int	unsigned long int
ll or LL	long long int unsigned long long int	long long int unsigned long long int
Both u or U , and ll or LL	unsigned long long int	unsigned long long int

Related References

“Compiler Command Line Options” on page 61

“langlvl” on page 175

longlong

► C ► C++

Purpose

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

Syntax

► — -q — longlong — nolonglong —

Default

The default with `xlc`, `xlC` and `cc` is `-qlonglong`, which defines `_LONG_LONG` (**long long** types will work in programs). The default with `c89` is `-qnolonglong` (**long long** types are not supported).

Notes

► C This option cannot be specified when the selected language level is `stdc99` or `extc99`. It is used to control the long long support that is provided as an extension to the C89 standard. This extension is slightly different from the long long support that is part of the C99 standard.

Examples

1. To compile `myprogram.c` so that **long long ints** are not allowed, enter:

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

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

```
xlc myprogram.c -D_LARGE_FILES -qlonglong
```

Related References

“Compiler Command Line Options” on page 61

M

► C ► C++

Purpose

Creates an output file that contains targets suitable for inclusion in a description file for the **make** command.

Syntax

►► -M ◀◀

Notes

The **-M** option is functionally identical to the **-qmakedep** option.

.u files are not **make** files; **.u** files must be edited before they can be used with the **make** command. For more information on this command, see your operating system documentation.

The output file contains a line for the input file and an entry for each include file. It has the general form:

```
file_name.o:file_name.c
file_name.o:include_file_name
```

Include files are listed according to the search order rules for the **#include** preprocessor directive, described in *Directory Search Sequence for Include Files Using Relative Path Names*. If the include file is not found, it is not added to the **.u** file.

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

Examples

If you do not specify the **-o** option, the output file generated by the **-M** option is created in the current directory. It has a **.u** suffix. For example, the command:

```
x1C -M person_years.c
```

produces the output file **person_years.u**.

A **.u** file is created for every input file with a **.c** or **.i** suffix. Output **.u** files are not created for any other files. For example, the command:

```
x1C -M conversion.c filter.c /lib/libm.a
```

produces two output files, **conversion.u** and **filter.u**, and an executable file as well. No **.u** file is created for the library.

If the current directory is not writable, no **.u** file is created. If you specify **-ofile_name** along with **-M**, the **.u** file is placed in the directory implied by **-ofile_name**. For example, for the following invocation:

```
x1C -M -c t.c -o /tmp/t.o
```

places the **.u** output file in **/tmp/t.u**.

Related References

“Compiler Command Line Options” on page 61

“makedep” on page 204

“o” on page 219

ma

► C

Purpose

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

Syntax

► — -ma ————— ◀

Notes

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

This option does not apply to C++ programs. In C++ programs, you must instead specify **#include <malloc.h>** to include the **alloca** function declaration.

Example

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

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

Related References

"Compiler Command Line Options" on page 61

"#pragma alloca" on page 300

macpstr

► C

Purpose

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

Syntax

► `-q` nomacpstr
macpstr ►

See also “#pragma options” on page 325.

Notes

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

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

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

```
"\pABC"
```

to:

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

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

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

Examples of Pascal String Literals: The compiler replaces trigraph sequences by the corresponding single-character representation. For example:

```
"??/p pascal string"
```

becomes:

```
"\p pascal string"
```

The following are examples of valid Pascal string literals:

ANSI Mode `"\p pascal string"`

Each instance of a new-line character and an immediately preceding backslash (\) character is deleted, splicing the physical source lines into logical ones. For example:

```
"\p pascal \  
string"
```

Two Pascal string literals are concatenated to form one Pascal string literal. For example:

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

or

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

becomes:

```
"\06ABCDEF"
```

For the macro `ADDQUOTES`:

```
#define ADDQUOTES (x) #x
```

where `x` is:

```
\p pascal string
```

or

```
\p pascal \  
string
```

becomes:

```
"\p pascal string"
```

Note however that:

```
ADDQUOTES(This is not a "\p pascal string")
```

becomes:

```
"This is not a \"\p pascal string\""
```

Extended Mode Is the same as ANSI mode, except the macro definition would be:

```
#define ADDQUOTES_Ext (x) "x"
```

where `x` is the same as in the ANSI example:

```
\p pascal string  
\p pascal \  
string
```

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

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

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

gives:

```
"ABCpDEF"
```

- A Pascal string literal cannot be concatenated with a **wide** string literal.
- The compiler truncates a Pascal string literal that is longer than 255 bytes (excluding the length byte and the terminating NULL) to 255 characters.

- The compiler ignores the `-qmacpstr` option if `-qmbcs` or `-qdbcs` is used, and issues a warning message.
- Because there is no Pascal-string-literal processing of wide strings, using the escape sequence `\p` in a wide string literal with the `-qmacpstr` option, generates a warning message and the escape sequence is ignored.
- The Pascal string literal is *not* a basic type different from other C or C++ string literals. After the processing of the Pascal string literal is complete, the resulting string is treated the same as all other strings. If the program passes a C string to a function that expects a Pascal string, or vice versa, the behavior is undefined.
- Concatenating two Pascal string literals, for example, `strcat()`, does not result in a Pascal string literal. However, as described above, two adjacent Pascal string literals can be concatenated to form one Pascal string literal in which the first byte is the length of the new string literal.
- Modifying any byte of the Pascal string literal after the processing has been completed does not alter the original length value in the first byte.
- No errors or warnings are issued when the bytes of the processed Pascal string literal are modified.
- Entering the characters:

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

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

Example

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

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

Related References

“Compiler Command Line Options” on page 61

“mbcs, dbcs” on page 209

“#pragma options” on page 325

maf

► C ► C++

Purpose

Specifies whether floating-point multiply-add instructions are to be generated. This option affects the precision of floating-point intermediate results.

Syntax

►► — -q —  —►►

See also “#pragma options” on page 325.

Notes

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

Related References

“Compiler Command Line Options” on page 61

“float” on page 131

“#pragma options” on page 325

makedep

C C++

Purpose

Creates an output file that contains targets suitable for inclusion in a description file for the **make** command.

Syntax

`-q—makedep`

Notes

The **-qmakedep** option is functionally identical to the **-M** option.

.u files are not **make** files; **.u** files must be edited before they can be used with the **make** command. For more information on this command, see your operating system documentation.

If you do not specify the **-o** option, the output file generated by the **-qmakedep** option is created in the current directory. It has a **.u** suffix. For example, the command:

```
x1C -qmakedep person_years.c
```

produces the output file **person_years.u**.

A **.u** file is created for every input file with a **.c** or **.i** suffix. Output **.u** files are not created for any other files. For example, the command:

```
x1C -qmakedep conversion.c filter.c /lib/libm.a
```

produces two output files, **conversion.u** and **filter.u** (and an executable file as well). No **.u** file is created for the library.

If the current directory is not writable, no **.u** file is created. If you specify **-ofile_name** along with **-qmakedep**, the **.u** file is placed in the directory implied by **-ofile_name**. For example, for the following invocation:

```
x1C -qmakedep -c t.c -o /tmp/t.o
```

places the **.u** output file in **/tmp/t.u**.

The output file contains a line for the input file and an entry for each include file. It has the general form:

```
file_name.o:file_name.c  
file_name.o:include_file_name
```

Include files are listed according to the search order rules for the **#include** preprocessor directive, described in “Directory Search Sequence for Include Files Using Relative Path Names” on page 33. If the include file is not found, it is not added to the **.u** file.

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

Related References

“Compiler Command Line Options” on page 61

“M” on page 198

"o" on page 219

"Directory Search Sequence for Include Files Using Relative Path Names" on page 33

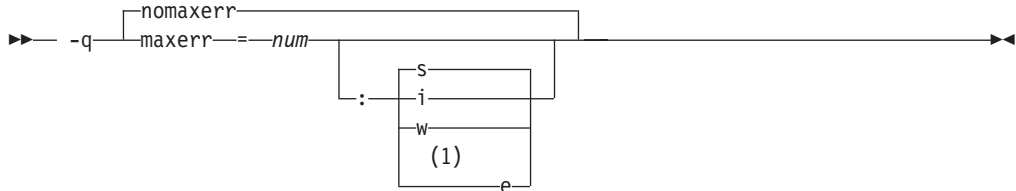
maxerr

C C++

Purpose

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

Syntax



Notes:

1 C only

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

<i>sev_level</i>	Description
i	Informational
w	Warning
e	Error (C only)
s	Severe error

Notes

If a severity level is not specified, the current value of the **-qhalt** option is used. The default value for **-qhalt** is **s** (severe error).

If the **-qmaxerr** option is specified more than once, the **-qmaxerr** option specified last determines the action of the option. If both the **-qmaxerr** and **-qhalt** options are specified, the **-qmaxerr** or **-qhalt** option specified last determines the severity level used by the **-qmaxerr** option.

An unrecoverable error occurs when the number of errors reached the limit specified. The error message issued is similar to:

```
1506-672 (U) The number of errors has reached the limit of ...
```

If **-qnomaxerr** is specified, the entire source file is compiled regardless of how many errors are encountered.

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

Examples

- To stop compilation of `myprogram.c` when 10 warnings are encountered, enter the command:

```
xlc myprogram.c -qmaxerr=10:w
```
- To stop compilation of `myprogram.c` when 5 severe errors are encountered, assuming that the current **-qhalt** option value is **S** (severe), enter the command:

```
x1C myprogram.c -qmaxerr=5
```

3. To stop compilation of myprogram.c when 3 informationals are encountered, enter the command:

```
x1C myprogram.c -qmaxerr=3:i
```

or:

```
x1C myprogram.c -qmaxerr=5:w qmaxerr=3 -qhalt=i
```

Related References

“Compiler Command Line Options” on page 61

“flag” on page 130

“halt” on page 143

“Message Severity Levels and Compiler Response” on page 379

maxmem

C C++

Purpose

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

Syntax

► `-qmaxmem=size` 8192 ◀

Notes

- A *size* value of -1 permits each optimization to take as much memory as it needs without checking for limits. Depending on the source file being compiled, the size of subprograms in the source, the machine configuration, and the workload on the system, this might exceed available system resources.
- The limit set by **-qmaxmem** is the amount of memory for specific optimizations, and not for the compiler as a whole. Tables required during the entire compilation process are not affected by or included in this limit.
- Setting a large limit has no negative effect on the compilation of source files when the compiler needs less memory.
- Limiting the scope of optimization does not necessarily mean that the resulting program will be slower, only that the compiler may finish before finding all opportunities to increase performance.
- Increasing the limit does not necessarily mean that the resulting program will be faster, only that the compiler is better able to find opportunities to increase performance if they exist.

Depending on the source file being compiled, the size of the subprograms in the source, the machine configuration, and the workload on the system, setting the limit too high might lead to page-space exhaustion. In particular, specifying **-qmaxmem=-1** allows the compiler to try and use an infinite amount of storage, which in the worst case can exhaust the resources of even the most well-equipped machine.

Example

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

```
x1C myprogram.c -qmaxmem=16384
```

Related References

“Compiler Command Line Options” on page 61

mbcs, dbcs

► C ► C++

Purpose

Use the **-qmbcs** option if your program contains multibyte characters. The **-qmbcs** option is equivalent to **-qdbcs**.

Syntax



See also “#pragma options” on page 325.

Notes

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

Example

To compile myprogram.c if it contains multibyte characters, enter:

```
x1C myprogram.c -qmbcs
```

Related References

“Compiler Command Line Options” on page 61

“#pragma options” on page 325

Appendix B, “National Languages Support in VisualAge C++” on page 401

mkshrojb

C C++

Purpose

Creates a shared object from generated object files.

Syntax

→ -qmkshrojb [=*priority*] →

where *priority* specifies the priority level for the file. *priority* may be any number from -214782623 (highest priority-initialized first) to 214783647 (lowest priority-initialized last). Numbers from -214783648 to -214782624 are reserved for system use. If no priority is specified the default priority of 0 is used. The priority is not used when linking shared objects (using the **xlc** command) written in C.

Notes

This option, together with the related options described below, should be used instead of the **makeC++SharedLib** command to create a shared object. The advantage to using this option is that the compiler will automatically include and compile the template instantiations in the tempinc directory.

The compiler will automatically export all global symbols from the shared object unless one explicitly specifies which symbols to export with the **-bE:**, **-bexport:** or **-bexpall** options.

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

The following related options can be used with the **-qmkshrojb** compiler option:

-oshared_file.o	Is the name of the file that will hold the shared file information. The default is shr.o.
-qexpfile=filename	Saves all exported symbols in <i>filename</i> . This option is ignored unless xlc automatically creates the export list.
-e name	Sets the entry name for the shared executable to name. The default is -bnoentry.

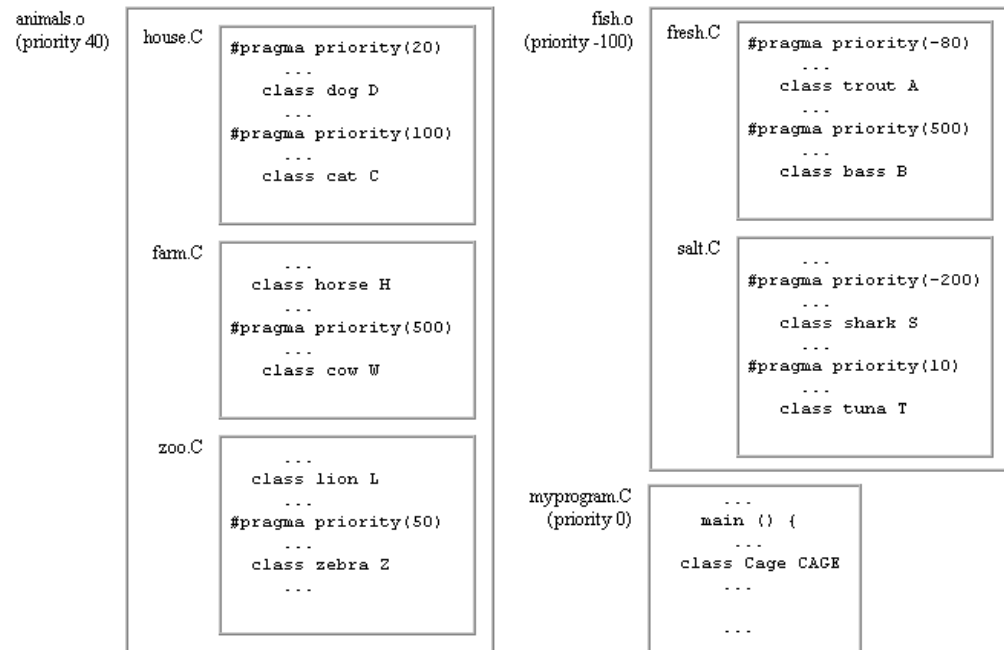
If you use **-qmkshrojb** to create a shared library, the compiler will:

1. If the user doesn't specify **-bE:**, **-bexport:**, **-bexpall** or **-bnoexpall**, create an export list containing all global symbols using the CreateExportList script. You can specify another script with the **-tE/-B** or **-qpath=E:** options.
2. If CreateExportList was used to create the export list and **-qexpfile** was specified, the export list is saved.
3. Calls the linker with the appropriate options and object files to build a shared object.

Example

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

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



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

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

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

To specify this initialization order, follow these steps:

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

```
x1C zoo.C -c -qpriority=-50
```
 - b. Compile the `house.C` and `farm.C` files without specifying the **-qpriority=N** option (so $N=0$) so objects within the files retain the priority numbers specified by their **#pragma priority(N)** directives:

```
x1C house.C farm.C -c
```
 - c. Combine these three files in a shared library. Use `x1C -qmkshrobj` to construct a library `animals.o` with a priority of 40:

```
x1C -qmkshrobj=40 -o animals.o house.o farm.o zoo.o
```
2. Develop an initialization order for the objects in `fresh.C`, and `salt.C`:
 - a. Compile the `fresh.C` and `salt.C` files:

```
x1C fresh.C salt.C -c
```

- b. To assure that all objects in fresh.C and salt.C are initialized before any other objects, use `x1C -qmkshrobj` to construct a library fish.o with a priority of -100.

```
x1C -qmkshrobj=-100 -o fish.o fresh.o salt.o
```

Because the shared library fish.o has a lower priority number (-100) than animals.o (40), when the files are placed in an archive file with the `ar` command, their objects are initialized first.

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

```
x1C myprogram.C -c
```

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

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

where:

`rv` Are two `ar` options. `r` replaces a named file if it already appears in the library, and `v` writes to standard output a file-by-file description of the making of the new library.

`libzoo.a` Is the name you specified for the archive file that will contain the shared object files and their priority levels.

`animals.o` `fish.o` Are the two shared files you created with `x1C -qmkshrobj`.

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

```
x1C -oanimal_time myprogram.o -L. -lzoo
```

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

Order of Initialization of Objects in libzoo.a			
File	Class Object	Priority Value	Comment
"fish.o"		-100	All objects in "fish.o" are initialized first because they are in a library prepared with <code>-qmkshrobj=-100</code> (lowest priority number, -100, specified for any files in this compilation)
	"shark S"	-100(-200)	Initialized first in "fish.o" because within file, <code>#pragma priority(-200)</code>
	"trout A"	-100(-80)	<code>#pragma priority(-80)</code>
	"tuna T"	-100(10)	<code>#pragma priority(10)</code>
	"bass B"	-100(500)	<code>#pragma priority(500)</code>
"myprog.o"		0	File generated with no priority specifications; default is 0
	"CAGE"	0(0)	Object generated in main with no priority specifications; default is 0

Order of Initialization of Objects in libzoo.a			
"animals.o"		40	File generated with -qmkshrobj=40
	"lion L"	40(-50)	Initialized first in file "animals.o" compiled with -qpriority=-50
	"horse H"	40(0)	Follows with priority of 0 (since -qpriority=N not specified at compilation and no #pragma priority(N) directive)
	"dog D"	40(20)	Next priority number (specified by #pragma priority(20))
	"zebra N"	40(50)	Next priority number from #pragma priority(50)
	"cat C"	40(100)	Next priority number from #pragma priority(100)
	"cow W"	40(500)	Next priority number from #pragma priority(500) (Initialized last)

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

Related References

"Compiler Command Line Options" on page 61

"b" on page 89

"e" on page 117

"expfile" on page 125

"o" on page 219

"path" on page 225

"priority" on page 232

"#pragma priority" on page 336

namemangling

C++

Purpose

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

Syntax



where available choices for mangling schemes are:

- ansi The name mangling scheme fully supports the various language features of Standard C++, including function template overloading.
- v5 The name mangling scheme is compatible with VisualAge C++ version 5.0.
- v4 The name mangling scheme is compatible with VisualAge C++ version 4.0.
- v3 Use this scheme for compatibility with link modules created with versions of VisualAge C++ released prior to version 4.0, or with link modules that were created with the **#pragma namemangling** or **-qnamemangling=compat** compiler options specified.

This scheme cannot be used when a function has the same name and the same function parameter list as a function template specialization. For example, VisualAge C++ will issue a diagnostic message in the following case when **-qnamemangling=compat** is enabled:

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

template int foo(T) { return 42; }

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

compat Same as the v3 suboption, described above.

See also “#pragma namemangling” on page 322.

Notes

By default VisualAge C++ uses a scheme that supports the C++ standard.

Related References

“Compiler Command Line Options” on page 61

“#pragma namemangling” on page 322

“#pragma nameManglingRule” on page 323

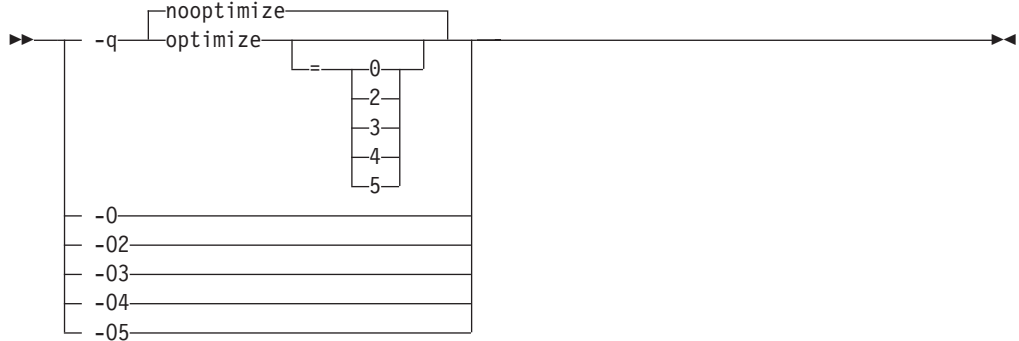
O, optimize

C C++

Purpose

Optimizes code at a choice of levels during compilation.

Syntax



where optimization settings are:

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

<p>-O3 -qOPTimize=3 (continued)</p>	<p>The aggressive optimizations performed when you specify -O3 are:</p> <ol style="list-style-type: none"> <p>Aggressive code motion, and scheduling on computations that have the potential to raise an exception, are allowed.</p> <p>Loads and floating-point computations fall into this category. This optimization is aggressive because it may place such instructions onto execution paths where they <i>will</i> be executed when they <i>may</i> not have been according to the actual semantics of the program.</p> <p>For example, a loop-invariant floating-point computation that is found on some, but not all, paths through a loop will not be moved at -O2 because the computation may cause an exception. At -O3, the compiler will move it because it is not certain to cause an exception. The same is true for motion of loads. Although a load through a pointer is never moved, loads off the static or stack base register are considered movable at -O3. Loads in general are not considered to be absolutely safe at -O2 because a program can contain a declaration of a static array <code>a</code> of 10 elements and load <code>a[60000000003]</code>, which could cause a segmentation violation.</p> <p>The same concepts apply to scheduling.</p> <p>Example:</p> <p>In the following example, at -O2, the computation of <code>b+c</code> is not moved out of the loop for two reasons:</p> <ol style="list-style-type: none"> it is considered dangerous because it is a floating-point operation it does not occur on every path through the loop <p>At -O3, the code is moved.</p> <pre> ... int i ; float a[100], b, c ; for (i = 0 ; i < 100 ; i++) { if (a[i] < a[i+1]) a[i] = b + c ; } ... </pre> <p>Conformance to IEEE rules are relaxed.</p> <p>With -O2 certain optimizations are not performed because they may produce an incorrect sign in cases with a zero result, and because they remove an arithmetic operation that may cause some type of floating-point exception.</p> <p>For example, <code>X + 0.0</code> is not folded to <code>X</code> because, under IEEE rules, <code>-0.0 + 0.0 = 0.0</code>, which is <code>-X</code>. In some other cases, some optimizations may perform optimizations that yield a zero result with the wrong sign. For example, <code>X - Y * Z</code> may result in a <code>-0.0</code> where the original computation would produce <code>0.0</code>.</p> <p>In most cases the difference in the results is not important to an application and -O3 allows these optimizations.</p> <p>Floating-point expressions may be rewritten.</p> <p>Computations such as <code>a*b*c</code> may be rewritten as <code>a*c*b</code> if, for example, an opportunity exists to get a common subexpression by such rearrangement. Replacing a divide with a multiply by the reciprocal is another example of reassociating floating-point computations.</p>
---------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

<p><code>-O3</code>, <code>-qOPTimize=3</code> (continued)</p>	<p>Notes</p> <ul style="list-style-type: none"> • <code>-qfloat=fltint:rsqrt</code> are on by default in <code>-O3</code>. • Built-in functions do not change <code>errno</code> at <code>-O3</code>. • Aggressive optimizations do <i>not</i> include the following floating-point suboptions: <code>-qfloat=hsflt</code>, <code>hssngl</code>, and <code>-qfloat=rndsngl</code>, or anything else that affects the precision mode of a program. • Integer divide instructions are considered too dangerous to optimize even at <code>-O3</code>. • The default <code>-qmaxmem</code> value is <code>-1</code> at <code>-O3</code>. • Refer to <code>-qflttrap</code> to see the behavior of the compiler when you specify <code>optimize</code> options with the <code>flttrap</code> option. • You can use the <code>-qstrict</code> and <code>-qstrict_induction</code> compiler options to turn off effects of <code>-O3</code> that might change the semantics of a program. Reference to the <code>-qstrict</code> compiler option can appear before or after the <code>-O3</code> option. • The <code>-O3</code> compiler option followed by the <code>-O</code> option leaves <code>-qignerrno</code> on.
<p><code>-O4</code> <code>-qOPTimize=4</code></p>	<p>This option is the same as <code>-O3</code>, except that it also:</p> <ul style="list-style-type: none"> • Sets the <code>-qipa</code> option • Sets the <code>-qhot</code> option • Sets the <code>-qarch</code> and <code>-qtune</code> options to the architecture of the compiling machine <p>Note: Later settings of <code>-O</code>, <code>-qcache</code>, <code>-qipa</code>, <code>-qarch</code>, and <code>-qtune</code> options will override the settings implied by the <code>-O4</code> option.</p>
<p><code>-O5</code> <code>-qOPTimize=5</code></p>	<p>This option is the same as <code>-O4</code>, except that it:</p> <ul style="list-style-type: none"> • Sets the <code>-qipa=level=2</code> option to perform full interprocedural data flow and alias analysis. <p>Note: Later settings of <code>-O</code>, <code>-qcache</code>, <code>-qipa</code>, <code>-qarch</code>, and <code>-qtune</code> options will override the settings implied by the <code>-O5</code> option.</p>
<p><code>-qNOOPTimize</code> <code>-qOPTimize=0</code></p>	<p>Performs only quick local optimizations such as constant folding and elimination of local common subexpressions.</p> <p>This setting implies <code>-qstrict_induction</code> unless <code>-qnostrict_induction</code> is explicitly specified.</p>

Notes

You can abbreviate `-qoptimize...` to `-qopt...`. For example, `-qnoopt` is equivalent to `-qnooptimize`.

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

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

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

Example

To compile myprogram.c for maximum optimization, enter:

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

Related References

“Compiler Command Line Options” on page 61

“arch” on page 83

“cache” on page 95

“float” on page 131

“g” on page 141

“ignprag” on page 153

“ipa” on page 163

“langlvl” on page 175

“strict” on page 261

“strict_induction” on page 262

“tune” on page 277



Purpose

Specifies an output location for the object, assembler, or executable files created by the compiler. When the **-o** option is used during compiler invocation, *file_spec* can be the name of either a file or a directory. When the **-o** option is used during direct linkage-editor invocation, *file_spec* can only be the name of a file.

Syntax

►— **-o**— *filespec*—————►

Notes

When **-o** is specified as part of a compiler invocation, *file_spec* can be the relative or absolute path name of either a directory or a file.

1. If *file_spec* is the name of a directory, files created by the compiler are placed into that directory.
2. If a directory with the name *file_spec* does not exist, the **-o** option specifies that the name of the file produced by the compiler will be *file_spec*. Otherwise, files created by the compiler will take on their default names. For example, the compiler invocation:

```
x1C test.c -c -o new.o
```

produces the object file **new.o** instead of **test.o**, and

```
x1C test.c -o new
```

produces the object file **new** instead of **a.out**.

A *file_spec* with a C or C++ source file suffix (**.C**, **.c**, or **.i**), such as *my_text.c* or *bob.i*, results in an error and neither the compiler nor the linkage editor is invoked.

If you use **-c** and **-o** together and the *filespec* does not specify a directory, you can only compile one source file at a time. In this case, if more than one source file name is listed in the compiler invocation, the compiler issues a warning message and ignores **-o**.

The **-E**, **-P**, and **-qsyntaxonly** options override the **-ofilename** option.

Example

To compile *myprogram.c* so that the resulting file is called **myaccount**, assuming that no directory with name **myaccount** exists, enter:

```
x1C myprogram.c -o myaccount
```

If the directory **myaccount** does exist, the executable file produced by the compiler is placed in the **myaccount** directory.

Related References

“Compiler Command Line Options” on page 61

“c” on page 94

“E” on page 115

“o”

“P” on page 222

“syntaxonly” on page 265

objmodel

C++

Purpose

Sets the type of object model.

Syntax

►► — -q—objmodel—=— compat
 ibm —————►►

where choices for object model are:

- | | |
|-------------------|------------------------------------------------------------------------------|
| -qobjmodel=compat | Uses the x1C object model compatible with previous versions of the compiler. |
| -qobjmodel=ibm | Uses the new object model. |

See also “#pragma object_model” on page 324.

Example

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

```
x1C myprogram.C -qobjmodel=ibm
```

Related Concepts

“Object Models” on page 4

Related References

“Compiler Command Line Options” on page 61

“#pragma object_model” on page 324

oldpassbyvalue

C++

Purpose

Specifies how classes containing const or reference members are passed in function arguments. All classes in the compilation unit are affected by this option.

Syntax

►► — -q — nooldpassbyvalue — oldpassbyvalue — ►►

See also “#pragma pass_by_value” on page 335.

Notes

The VisualAge C++ v3.6 compiler only uses pass by value if the class has no const or reference data member and the copy constructor is trivial and the destructor is trivial. The VisualAge C++ v5.0 compiler uses pass by value if the copy constructor is trivial and the destructor is trivial, regardless of const or reference data members.

When **-qoldpassbyvalue** is specified, the compiler mimics the VisualAge C++ v3.6 compiler in that when a class containing a const or reference member is passed as a function argument, it is not passed by value. All such classes in the compilation unit are affected.

The **#pragma pass_by_value** directive overrides **-qoldpassbyvalue**, and gives you additional control in enabling this options. See the description for **#pragma pass_by_value** for more information.

Related References

“Compiler Command Line Options” on page 61

“#pragma pass_by_value” on page 335

P

► C ► C++

Purpose

Preprocesses the C or C++ source files named in the compiler invocation and creates an output preprocessed source file, *file_name.i* for each input source file *file_name.c* or *file_name.C*. The **-P** option calls the preprocessor directly as `/usr/vac/exe/xlCcpp`.

Syntax

►► -P ◀◀

Notes

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

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

#line directives are not issued.

The **-P** option cannot accept a preprocessed source file, *file_name.ias* input. Source files with unrecognized filename suffixes are treated and preprocessed as C files, and no error message is generated.

In extended mode, the preprocessor interprets the backslash character when it is followed by a new-line character as line-continuation in:

- macro replacement text
- macro arguments
- comments that are on the same line as a preprocessor directive.

Line continuations elsewhere are processed in **ANSI** mode only.

The **-P** option is overridden by the **-E** option. The **-P** option overrides the **-c**, **-o**, and **-qsyntaxonly** option. The **-C** option may be used in conjunction with both the **-E** and **-P** options.

The default is to compile and link-edit C or C++ source files to produce an executable file.

Related References

“Compiler Command Line Options” on page 61

“C” on page 93

“c” on page 94

“E” on page 115

“o” on page 219

p

► C ► C++

Purpose

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

Syntax

►► -p ◀◀

Notes

If the `-qtbtable` option is not set, the `-p` option will generate full traceback tables.

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

Example

To compile `myprogram.c` so that it can be used with the operating system `prof` command, enter:

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

Related References

“Compiler Command Line Options” on page 61

“tbtable” on page 268

Also, on the Web see:

`prof` Command section in *Commands Reference, Volume 4: n through r* for information about profiling.

pascal

► c

Purpose

Ignores the word **pascal** in type specifiers and function declarations.

Syntax

►► -q nopascal
 pascal ◀◀

Notes

This option can be used to improve compatibility of IBM VisualAge C++ programs on some other systems.

Related References

“Compiler Command Line Options” on page 61

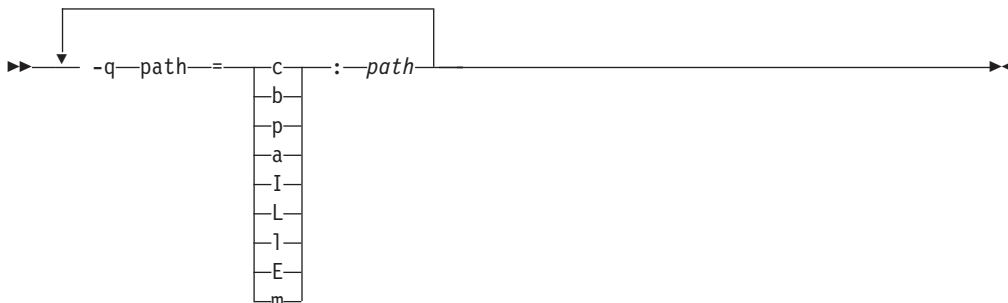
path

C C++

Purpose

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

Syntax



where program names are:

Program	Description
c	Compiler front end
b	Compiler back end
p	Compiler preprocessor
a	Assembler
I	Interprocedural Analysis tool - compile phase
L	Interprocedural Analysis tool - link phase
l	Linkage editor
E	CreateExportList utility
m	Linkage helper (munch utility)

Notes

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

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

Examples

To compile myprogram.c using a substitute x1C compiler in **/lib/tmp/mine/** enter:

```
x1C myprogram.c -qpath=c:/lib/tmp/mine/
```

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

```
x1C myprogram.c -qpath=l:/lib/tmp/mine/
```

Related References

“Compiler Command Line Options” on page 61

“B” on page 88

“F” on page 127

“t” on page 266

pdf1, pdf2

C C++

Purpose

Tunes optimizations through *profile-directed feedback* (PDF), where results from sample program execution are used to improve optimization near conditional branches and in frequently executed code sections.

Syntax



Notes

To use PDF, follow these steps:

1. Compile some or all of the source files in a program with the **-qpdf1** option. You need to specify the **-O2** option, or preferably the **-O3**, **-O4**, or **-O5** option, for optimization. Pay special attention to the compiler options that you use to compile the files, because you will need to use the same options later.
In a large application, concentrate on those areas of the code that can benefit most from optimization. You do not need to compile all of the application's code with the **-qpdf1** option.
2. Run the program all the way through using a typical data set. The program records profiling information when it finishes. You can run the program multiple times with different data sets, and the profiling information is accumulated to provide an accurate count of how often branches are taken and blocks of code are executed.
Important: Use data that is representative of the data that will be used during a normal run of your finished program.
3. Relink your program using the same compiler options as before, but change **-qpdf1** to **-qpdf2**. Remember that **-L**, **-l**, and some others are linker options, and you can change them at this point. In this second compilation, the accumulated profiling information is used to fine-tune the optimizations. The resulting program contains no profiling overhead and runs at full speed.

For best performance, use the **-O3**, **-O4**, or **-O5** option with all compilations when you use PDF.

The profile is placed in the current working directory or in the directory that the PDFDIR environment variable names, if that variable is set.

To avoid wasting compilation and execution time, make sure that the PDFDIR environment variable is set to an absolute path. Otherwise, you might run the application from the wrong directory, and it will not be able to locate the profile data files. When that happens, the program may not be optimized correctly or may be stopped by a segmentation fault. A segmentation fault might also happen if you change the value of the PDFDIR variable and execute the application before finishing the PDF process.

Because this option requires compiling the entire application twice, it is intended to be used after other debugging and tuning is finished, as one of the last steps before putting the application into production.

Restrictions

- PDF optimizations require at least the **-O2** optimization level.
- You must compile the main program with PDF for profiling information to be collected at run time.
- Do not compile or run two different applications that use the same PDFDIR directory at the same time, unless you have used the **-qipa=pdfname** suboption to distinguish the sets of profiling information.
- You must use the same set of compiler options at all compilation steps for a particular program. Otherwise, PDF cannot optimize your program correctly and may even slow it down. All compiler settings must be the same, including any supplied by configuration files.
- Avoid mixing PDF files created by the current version of VisualAge C++ with PDF files created by other versions of the compiler.
- If **-qipa** is not invoked either directly or through other options, **-qpdf1** and **-qpdf2** will invoke the **-qipa=level=0** option.
- If you do compile a program with **-qpdf1**, remember that it will generate profiling information when it runs, which involves some performance overhead. This overhead goes away when you recompile with **-qpdf2** or with no PDF at all.

The following commands, found in **/usr/xlopt/bin**, are available for managing the PDFDIR directory:

`resetpdf [pathname]` Sets to zeros all profiling information (but does not remove the data files) from the *pathname* directory, or from the PDFDIR directory if *pathname* is not specified, or from the current directory if PDFDIR is not set.

When you make changes to the application and recompile some files, the profiling information for those files is automatically reset because the changes may alter the program flow. Run **resetpdf** to reset the profiling information for the entire application after you make significant changes that may change execution counts for parts of the program that were not recompiled.

`cleanpdf [pathname]` Removes all profiling information from the *pathname* directory; or if *pathname* is not specified, from the PDFDIR directory; or if PDFDIR is not set, from the current directory.

Removing the profiling information reduces the runtime overhead if you change the program and then go through the PDF process again.

Run this program after compiling with **-qpdf2**, or after finishing with the PDF process for a particular application. If you continue using PDF with an application after running **cleanpdf**, you must recompile all the files with **-qpdf1**.

Examples

Here is a simple example:

```
/* Set the PDFDIR variable. */
export PDFDIR=$HOME/project_dir

/* Compile all files with -qpdf1. */
x1C -qpdf1 -O3 file1.C file2.C file3.C

/* Run with one set of input data. */
a.out <sample.data

/* Recompile all files with -qpdf2. */
x1C -qpdf2 -O3 file1.C file2.C file3.C

/* The program should now run faster than
   without PDF if #the sample data is typical. */
```

Here is a more elaborate example.

```
/* Set the PDFDIR variable. */
export PDFDIR=$HOME/project_dir

/* Compile most of the files with -qpdf1. */
x1C -qpdf1 -O3 -c file1.C file2.C file3.C

/* This file is not so important to optimize.
x1C -c file4.C

/* Non-PDF object files such as file4.o can be linked in. */
x1C -qpdf1 file1.o file2.o file3.o file4.o

/* Run several times with different input data. */
a.out <polar_orbit.data
a.out <elliptical_orbit.data
a.out <geosynchronous_orbit.data

/* No need to recompile the source of non-PDF object files (file4.C). */
x1C -qpdf2 -O3 file1.C file2.C file3.C

/* Link all the object files into the final application. */
x1C file1.o file2.o file3.o file4.o
```

Related References

“Compiler Command Line Options” on page 61

“O, optimize” on page 215

pg

► C ► C++

Purpose

Sets up the object files for profiling, but provides more information than is provided by the `-p` option.

If the `-qtbtable` option is not set, the `-pg` option will generate full traceback tables.

Syntax

►— `-pg` —◄

Example

To compile `myprogram.c` for use with the AIX `gprof` command, enter:

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

Remember to compile *and* link with the `-pg` option. For example:

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

Related References

“Compiler Command Line Options” on page 61

“tbtable” on page 268

Also, on the Web see:

`gprof` Command section in *Commands Reference, Volume 2: d through h* for information about profiling.

phsinfo

> C > C++

Purpose

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

Syntax

► — -q — nophsinfo
phsinfo —————►

Notes

The output takes the form *number1* | *number2* for each phase where *number1* represents the CPU time used by the compiler and *number2* represents the total of the compiler time and the time that the CPU spends handling system calls.

Example

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

```
x1C myprogram.C -qphsinfo
```

Related References

“Compiler Command Line Options” on page 61

print

> C > C++

Purpose

Suppresses listings. **-qnoprint** overrides all of the listing-producing options, regardless of where they are specified.

Syntax



Notes

The default is to not suppress listings if they are requested.

The options that produce listings are:

- `-qattr`
- `-qlist`
- `-qlistopt`
- `-qsource`
- `-qxref`

Example

To compile `myprogram.c` and suppress all listings, even if some files have `#pragma options source` and similar directives, enter:

```
x1C myprogram.c -qnoprint
```

Related References

“Compiler Command Line Options” on page 61

“attr” on page 87

“list” on page 193

“listopt” on page 194

“source” on page 254

“xref” on page 294

priority

C++

Purpose

Specifies the priority level for the initialization of static constructors

Syntax

►► `-qpriority=number` ◀◀

See also “`#pragma priority`” on page 336 and “`#pragma options`” on page 325.

Notes

number Is the initialization priority level assigned to the static constructors within a file, or the priority level of a shared or non-shared file or library.

You can specify a priority level from $-(2147483647 + 1)$ (highest priority) to $+2147483647$ (lowest priority).

Example

To compile the file `myprogram.C` to produce an object file `myprogram.o` so that objects within that file have an initialization priority of `-200`, enter:

```
x1C myprogram.C -c -qpriority=-200
```

All objects in the resulting object file will be given an initialization priority of `-200`, provided that the source file contains no `#pragma priority(number)` directives specifying a different priority level.

Related References

“Compiler Command Line Options” on page 61

“`#pragma options`” on page 325

“`#pragma priority`” on page 336

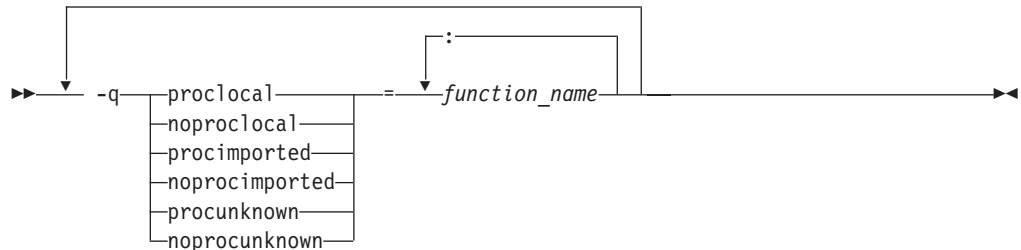
procllocal, procimported, procunknown

C C++

Purpose

Marks functions as local, imported, or unknown.

Syntax



See also “#pragma options” on page 325.

Default

The default is to assume that all functions whose definition is in the current compilation unit are local **procllocal**, and that all other functions are unknown **procunknown**. If any functions that are marked as local resolve to shared library functions, the linkage editor will detect the error and issue warnings such as:

```
ld: 0711-768 WARNING: Object foo.o, section 1, function .printf:
The branch at address 0x18 is not followed by a recognized no-op
or TOC-reload instruction. The unrecognized instruction is 0x83E1004C.
```

An executable file is produced, but it will not run. The error message indicates that a call to **printf** in object file **foo.o** caused the problem. When you have confirmed that the called routine should be imported from a shared object, recompile the source file that caused the warning and explicitly mark **printf** as imported. For example:

```
xlc -c -qprocimported=printf foo.c
```

Notes

Local functions Are statically bound with the functions that call them. **-qprocllocal** changes the default to assume that all functions are local. **-qprocllocal=names** marks the named functions as local, where *names* is a list of function identifiers separated by colons (:). The default is not changed.

Imported functions Smaller, faster code is generated for calls to functions marked as local. Are dynamically bound with a shared portion of a library. **-qprocimported** changes the default to assume that all functions are imported. **-qprocimported=names** marks the named functions as imported, where *names* is a list of function identifiers separated by colons (:). The default is not changed.

The code generated for calls to functions marked as imported might be larger, but it is faster than the default code sequence generated for functions marked as unknown. If any marked functions are resolved to statically bound objects, the generated code may be larger and run more slowly than the default code sequence generated for unknown functions.

Unknown functions Are resolved to either statically or dynamically bound objects during link-editing. **-qprocunknown** changes the default to assume that all functions are unknown. **-qprocunknown=names** marks the named functions as unknown, where *names* is a list of function identifiers separated by colons (:). The default is not changed.

Conflicts among the procedure-marking options are resolved in the following manner:

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

Example

To compile myprogram.c along with the archive library **oldprogs.a** so that:

- functions **fun** and **sun** are specified as **local**,
- functions **moon** and **stars** are specified as **imported**, and,
- function **venus** is specified as **unknown**,

enter:

```
x1C myprogram.c oldprogs.a -qprolocal=fun(int):sun()  
-qprocimported=moon():stars(float) -qprocunknown=venus()
```

Related References

“Compiler Command Line Options” on page 61

proto

► C

Purpose

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

Syntax

► — -q — no proto —

Notes

This option asserts that procedure call points agree with their declarations even if the procedure has not been prototyped. Callers can pass floating-point arguments in floating-point registers only and not in General-Purpose Registers (GPRs). The compiler assumes that the arguments on procedure calls are the same types as the corresponding parameters of the procedure definition.

You can obtain warnings for functions that do not have prototypes.

Example

To compile `my_c_program.c` to assume that all functions are prototyped, enter:

```
xlc my_c_program.c -qproto
```

Related References

“Compiler Command Line Options” on page 61

Q

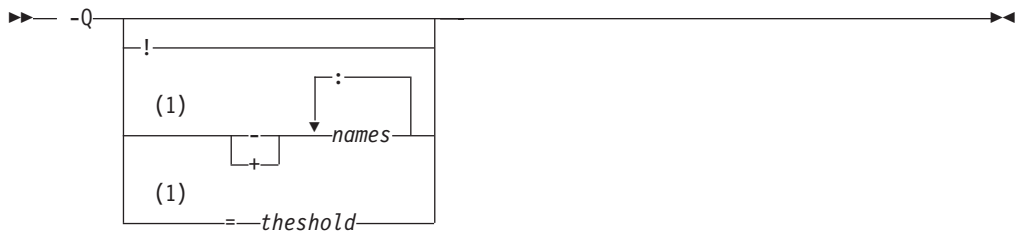
► C ► C++

Purpose

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

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

Syntax



Notes:

1 C only

► C++ In the C++ language, the following -Q options apply:

- Q Compiler inlines all functions that it can.
- Q! Compiler does not inline any functions.

► C In the C language, the following -Q options apply:

- Q Attempts to inline all appropriate functions with 20 executable source statements or fewer, subject to the setting of any of the suboptions to the -Q option. If -Q is specified last, all functions are inlined.
- Q! Does not inline any functions. If -Q! is specified last, no functions are inlined.
- Q-names Does not inline functions listed by *function_name*. Separate each *function_name* with a colon (:). All other appropriate functions are inlined. The option implies -Q.

For example:

```
-Q-salary:taxes:expenses:benefits
```

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

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

-Q+names Attempts to inline the functions listed by *function_name* and any other appropriate functions. Each *function_name* must be separated by a colon (:). The option implies **-Q**.

For example,

```
-Q+food:clothes:vacation
```

causes all functions named **food**, **clothes**, or **vacation** to be inlined if possible, along with any other functions eligible for inlining.

A warning message is issued for functions that are not defined in the source file or that are defined but cannot be inlined.

This suboption overrides any setting of the *threshold* value. You can use a threshold value of zero along with **-Q+function_name** to inline specific functions. For example:

```
-Q=0
```

followed by:

```
-Q+salary:taxes:benefits
```

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

-Q=threshold Sets a size limit on the functions to be inlined. The number of executable statements must be less than or equal to *threshold* for the function to be inlined. *threshold* must be a positive integer. The default value is 20. Specifying a threshold value of **0** causes no functions to be inlined except those functions marked with the **__inline**, **_Inline**, or **_inline** keywords.

The *threshold* value applies to logical C statements. Declarations are not counted, as you can see in the example below:

```
increment()
{
  int a, b, i;
  for (i=0; i<10; i++) /* statement 1 */
  {
    a=i;             /* statement 2 */
    b=i;             /* statement 3 */
  }
}
```

Default

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

- If you specify the **-g** option (to generate debug information), no functions are inlined.
- If you optimize your programs, (specify the **-O** option) the compiler attempts to inline the functions declared as inline.

Notes

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

Because inlining does not always improve run time, you should test the effects of this option on your code.

Do not attempt to inline recursive or mutually recursive functions.

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

The **inline**, **_inline**, **_Inline**, and **__inline** language keywords override all **-Q** options except **-Q!**. The compiler will try to inline functions marked with these keywords regardless of other **-Q** option settings.

To maximize inlining:

- for C programs, specify optimization (**-O**) and also specify the appropriate **-Q** options for the C language.
- for C++ programs, specify optimization (**-O**) but do not specify the **-Q** option.

Examples

To compile the program `myprogram.c` so that no functions are inlined, enter:

```
x1C myprogram.c -O -Q!
```

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

```
x1C my_c_program.c -O -Q=12
```

Related References

“Compiler Command Line Options” on page 61

“inline” on page 159

“O, optimize” on page 215

“Q” on page 236

“The inline, _Inline, _inline, and __inline Function Specifiers” on page 161

r

> C > C++

Purpose

Produces a relocatable object. This permits the output file to be produced even though it contains unresolved symbols.

Syntax

▶▶ -r ◀◀

Notes

A file produced with this flag is expected to be used as a file parameter in another call to x1C.

Example

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

```
x1C myprogram.c myprog2.c -r -o mytest.o
```

Related References

“Compiler Command Line Options” on page 61

report

> C > C++

Purpose

Instructs the compiler to produce transformation reports that show how program loops are parallelized and/or optimized. The transformation reports are included as part of the compiler listing.

Syntax

► — -q — noreport
report —————►

Notes

Specifying **-qreport** together with **-qhot** instructs the compiler to produce a pseudo-C code listing and summary showing how loops are transformed. You can use this information to tune the performance of loops in your program.

Specifying **-qreport** together with **-qsmp** instructs the compiler to also produce a report showing how the program deals with data and automatic parallelization of loops in your program. You can use this information to determine how loops in your program are or are not parallelized.

The pseudo-C code listing is not intended to be compilable. Do not include any of the pseudo-C code in your program, and do not explicitly call any of the internal routines whose names may appear in the pseudo-C code listing.

Example

To compile `myprogram.c` so the compiler listing includes a report showing how loops are optimized, enter:

```
xlc -qhot -O3 -qreport myprogram.c
```

To compile `myprogram.c` so the compiler listing also includes a report showing how parallelized loops are transformed, enter:

```
xlc -qsmp -O3 -qreport myprogram.c
```

Related References

“Compiler Command Line Options” on page 61

“hot” on page 146

“smp” on page 252

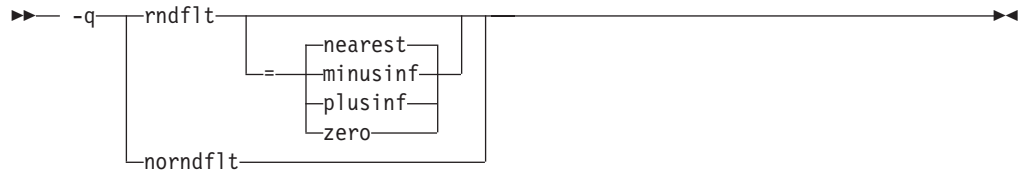
rndflt

C C++

Purpose

This option controls the compile-time rounding mode of constant floating point expressions. It does not affect run-time rounding.

Syntax



where available rounding options are:

Option	Effect
nearest	Round to nearest representable number. This is the default.
minusinf	Round toward minus infinity.
plusinf	Round toward plus infinity.
zero	Round toward zero.

Notes

By default, constant floating-point expressions are rounded toward the nearest representable number at compile time.

The following table describes the effect of specifying `-qrndflt=option` for each of the following options.

Compile-time floating-point arithmetic can have two effects on program results:

- In specific cases, the result of a computation at compile time might differ slightly from the result that would have been calculated at run time. The reason is that more rounding operations occur at compile time. For example, where a multiply-add floating point operation might be used at run time, separate multiply and add operations might be used at compile time, producing a slightly different result.
- Computations that produce exceptions can be folded to the IEEE result that would have been produced by default in a run-time operation. This would prevent an exception from occurring at run time. The `-qfltrp` option can be used to generate instructions that detect and trap floating-point exceptions.

In general, code that affects the rounding mode at run time should be compiled with the option that matches that rounding mode. For example, when the following program is compiled, the expression `1.0/3.0` is folded at compile time into a double-precision result:

```
main()
{
    float x, y;
    int i;
    x = 1.0/3.0;
    i = *(int *)&x;
    printf("1/3 = %.8x\n", i);
}
```

```
x = 1.0;
y = 3.0;
x = x/y;
i = *(int *)&x;
printf("1/3 = %.8x\n", i);
}
```

This result is then converted to single precision and stored in float x.

The **-qfloat=nofold** option can be specified to suppress all compile-time folding of floating-point computations. For example, the following code fragment may be evaluated either at compile time or at run time, depending on the setting of **-qfloat** and other options:

```
x = 1.0;
y = 3.0;
x = x/y;
```

The **-qrndflt** option only affects compile-time rounding of floating-point computations. If this code is evaluated at run time, the default run-time rounding of "round to nearest" is still in effect and takes precedence over the compile-time rounding mode.

Related References

"Compiler Command Line Options" on page 61

"float" on page 131

"flttrap" on page 135

"rndflt" on page 241

rndsngl

► C ► C++

Purpose

Specifies that the results of each single-precision **float** operation is to be rounded to single precision. **-qnorndsngl** specifies that rounding to single-precision happens only after full expressions have been evaluated.

Syntax

► -q norndsngl rndsngl ◀

See also “#pragma options” on page 325.

Notes

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

The **-qhsflt** option overrides the **-qrndsngl** options.

The **-qrndsngl** option is intended for specific applications in which floating-point computations have known characteristics. Using this option when compiling other application programs can produce incorrect results without warning.

Related References

“Compiler Command Line Options” on page 61

“float” on page 131

“hsflt” on page 148

“#pragma options” on page 325

ro

> C > C++

Purpose

Specifies the storage type for string literals.

Syntax

► — -q —  —►

See also “#pragma options” on page 325.

Default

The default with **xlc**, **x1C** and **c89** is **-qro**. The default with **cc** is **-qnoro**.

Notes

If **-qro** is specified, the compiler places string literals in read-only storage. If **-qnoro** is specified, string literals are placed in read/write storage.

You can also specify the storage type in your source program using:

```
#pragma strings storage_type
```

where *storage_type* is **read-only** or **writable**.

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

Example

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

```
x1C myprogram.c -qnoro
```

Related References

“Compiler Command Line Options” on page 61

“#pragma options” on page 325

roconst

► C ► C++

Purpose

Specifies the storage location for constant values.

Syntax

► -q roconst
noroconst ◀

See also “#pragma options” on page 325.

Default

The default with `xlc`, `x1C` and `c89` is `-qroconst`. The default with `cc` is `-qnoroconst`.

Notes

If `-qroconst` is specified, the compiler places constants in read-only storage. If `-qnoroconst` is specified, constant values are placed in read/write storage.

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

Constant value in the context of the `-qroconst` option refers to variables that are qualified by `const` (including `const`-qualified characters, integers, floats, enumerations, structures, unions, and arrays). The following variables do not apply to this option:

- variables qualified with `volatile` and aggregates (such as a `struct` or a `union`) that contain `volatile` variables
- pointers and complex aggregates containing pointer members
- automatic and static types with block scope
- uninitialized types
- regular structures with all members qualified by `const`
- initializers that are addresses, or initializers that are cast to non-address values

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

Related References

“Compiler Command Line Options” on page 61

“ro” on page 244

“#pragma options” on page 325

rrm

> C > C++

Purpose

Prevents floating-point optimizations that are incompatible with run-time rounding to plus and minus infinity modes.

Syntax

►► -q norr
rrm ◄◄

See also “#pragma options” on page 325.

Notes

This option informs the compiler that, at run time, the floating-point rounding mode may change or that the mode is not set to **-yn** (rounding to the nearest representable number.)

-qrrm must also be specified if the Floating Point Status and Control register is changed at run time.

The default, **-qnorr**, generates code that is compatible with run-time rounding modes **nearest** and **zero**. For a list of rounding mode options, see the **-y** compiler option.

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

Related References

“Compiler Command Line Options” on page 61

“float” on page 131

“#pragma options” on page 325

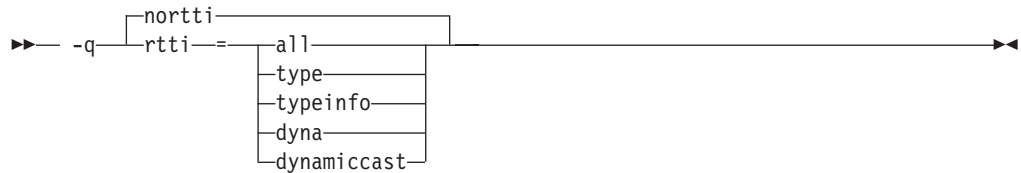
rtti

C++

Purpose

Use this option to generate run-time type identification (RTTI) information for the typeid operator and the dynamic_cast operator.

Syntax



where available suboptions are:

all	The compiler generates the information needed for the RTTI typeid and dynamic_cast operators. If you specify just -qrtti, this is the default suboption.
type typeinfo	The compiler generates the information needed for the RTTI typeid operator, but the information needed for dynamic_cast operator is not generated.
dyna dynamiccast	The compiler generates the information needed for the RTTI dynamic_cast operator, but the information needed for typeid operator is not generated.

Notes

For best run-time performance, suppress RTTI information generation with the default **-qnortti** setting.

The C++ language offers a (RTTI) mechanism for determining the class of an object at run time. It consists of two operators:

- one for determining the run-time type of an object (typeid), and,
- one for doing type conversions that are checked at run time (dynamic_cast).

A type_info class describes the RTTI available and defines the type returned by the typeid operator.

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

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

Related References

“Compiler Command Line Options” on page 61

S

► C ► C++

Purpose

This option strips the symbol table, line number information, and relocation information from the output file. Specifying `-s` saves space, but limits the usefulness of traditional debug programs when you are generating debug information using options such as `-g`.

Syntax

► `-s` ◀

Notes

Using the `strip` command has the same effect.

Related References

“Compiler Command Line Options” on page 61

“g” on page 141

showinc

> C > C++

Purpose

If used with the **-qsource** compiler option, all include files are shown in the source listing.

Syntax

► — -q — noshowinc
showinc —————►

See also “#pragma options” on page 325.

Example

To compile myprogram.c so that all included files appear in the source listing, enter:

```
x1C myprogram.c -qsource -qshowinc
```

Related References

“Compiler Command Line Options” on page 61

“source” on page 254

“#pragma options” on page 325

smallstack

► C ► C++

Purpose

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

Syntax

► — -q —  —

Notes

AIX limits the stack size to 256 MB. Programs that allocate large amounts of data to the stack may result in stack overflows. This option can reduce the stack frame to help avoid overflows.

This option is only valid when used together with IPA (**-qipa**, **-O4**, **-O5** compiler options).

Specifying this option may adversely affect program performance.

Example

To compile myprogram.c to use a small stack frame, enter:

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

Related References

“Compiler Command Line Options” on page 61

“g” on page 141

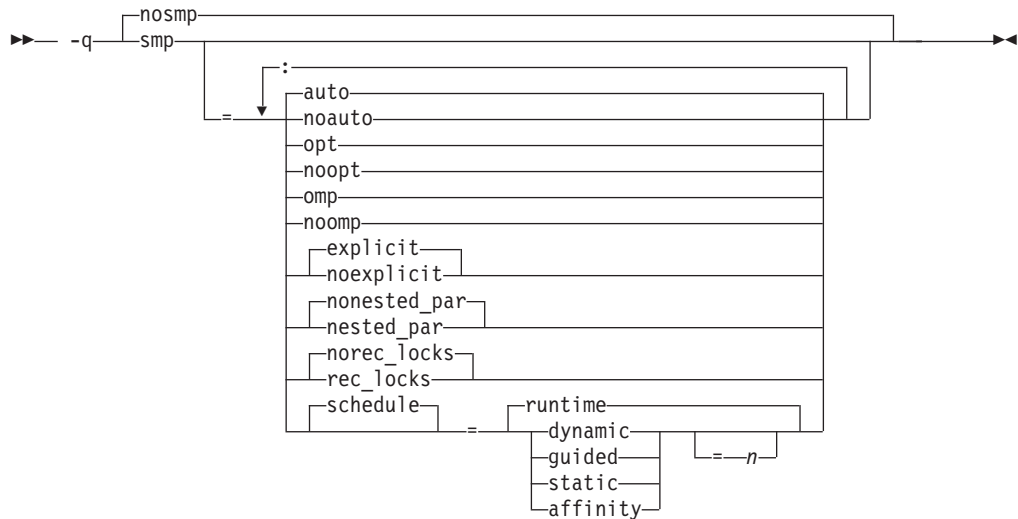
smp

C C++

Purpose

Enables automatic parallelization of program code.

Syntax



where:

- | | |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| auto | Enables automatic parallelization and optimization of program code. |
| noauto | Disables automatic parallelization of program code. Program code explicitly parallelized with SMP or OMP pragma statements is optimized. |
| opt | Enables automatic parallelization and optimization of program code. |
| noopt | Enables automatic parallelization, but disables optimization of parallelized program code. Use this setting when debugging parallelized program code.. |
| omp | Enables strict compliance to the OMP standard. Automatic parallelization is disabled. Parallelized program code is optimized. Only OMP parallelization pragmas are recognized. |
| noomp | Enables automatic parallelization and optimization of program code. |
| explicit | Enables pragmas controlling explicit parallelization of loops. |
| noexplicit | Disables pragmas controlling explicit parallelization of loops. |

nested_par	If specified, nested parallel constructs are not serialized. nested_par does not provide true nested parallelism because it does not cause new team of threads to be created for nested parallel regions. Instead, threads that are currently available are re-used.
	This option should be used with caution. Depending on the number of threads available and the amount of work in an outer loop, inner loops could be executed sequentially even if this option is in effect. Parallelization overhead may not necessarily be offset by program performance gains.
nonested_par	Disables parallelization of nested parallel constructs.
rec_locks	If specified, recursive locks are used, and nested critical sections will not cause a deadlock.
norec_locks	If specified, recursive locks are not used.
schedule=sched_type[=n]	Specifies what kind of scheduling algorithms and chunking are used for loops to which no other scheduling algorithm has been explicitly assigned in the source code. If <i>sched_type</i> is not specified, runtime is assumed for the default setting.

Notes

- The **-qnosmp** default option setting specifies that no code should be generated for parallelization directives, though syntax checking will still be performed. Use **-qignprag=omp:ibm** to completely ignore parallelization directives.
- Specifying **-qsmp** without suboptions is equivalent to specifying **-qsmp=auto:explicit:noomp:norec_locks:nonested_par:schedule=runtime** or **-qsmp=opt:explicit:noomp:norec_locks:nonested_par:schedule=runtime**.
- Specifying **-qsmp** implicitly sets **-O2**. The **-qsmp** option overrides **-qnooptimize**, but does not override **-O3**, **-O4**, or **-O5**. When debugging parallelized program code, you can disable optimization in parallelized program code by specifying **qsmp=noopt**.
- Specifying **-qsmp** defines the **_IBMSMP** preprocessing macro.
- **-qsmp** must be used only with thread-safe compiler mode invocations such as **xlc_r**. These invocations ensure that the **pthreads**, **xlsmp**, and thread-safe versions of all default run-time libraries are linked to the resulting executable.

Related Concepts

“Program Parallelization” on page 9

Related Tasks

“Set Parallel Processing Run-time Options” on page 20

“Control Parallel Processing with Pragmas” on page 45

Related References

“Compiler Command Line Options” on page 61

“O, optimize” on page 215

“threaded” on page 273

“Pragmas to Control Parallel Processing” on page 344

“#pragma ibm schedule” on page 352

“IBM SMP Run-time Options for Parallel Processing” on page 383

“OpenMP Run-time Options for Parallel Processing” on page 386

“Built-in Functions Used for Parallel Processing” on page 388

source

> C > C++

Purpose

Produces a compiler listing and includes source code.

Syntax

► — -q —  —►

See also “#pragma options” on page 325.

Notes

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

Parts of the source can be selectively printed by using pairs of **#pragma options source** and **#pragma options nosource** preprocessor directives throughout your source program. The source following **#pragma options source** and preceding **#pragma options nosource** is printed.

Examples

The following code causes the parts of the source code between the **#pragma options** directives to be included in the compiler listing:

```
#pragma options source
. . .
/* Source code to be included in the compiler listing
   is bracketed by #pragma options directives.
*/
. . .
#pragma options nosource
```

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

```
x1C myprogram.c -qsource
```

Related References

“Compiler Command Line Options” on page 61

“print” on page 231

“#pragma options” on page 325

spill

> C > C++

Purpose

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

Syntax

►► -q-spill=512
size►►

See also “#pragma options” on page 325.

Notes

If your program is very complex, or if there are too many computations to hold in registers at one time and your program needs temporary storage, you might need to increase this area. Do not enlarge the spill area unless the compiler issues a message requesting a larger spill area. In case of a conflict, the largest spill area specified is used.

Example

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

```
x1C myprogram.c -qspill=900
```

Related References

“Compiler Command Line Options” on page 61

“#pragma options” on page 325

spnans

► C ► C++

Purpose

Generates extra instructions to detect signalling NaN on conversion from single precision to double precision. The **-qnospnans** option specifies that this conversion need not be detected.

Syntax

► -q nospnans spnans ◀

See “#pragma options” on page 325.

Notes

The **-qhsflt** option overrides the **-qspnans** option

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

Related References

“Compiler Command Line Options” on page 61

“float” on page 131

“hsflt” on page 148

“#pragma options” on page 325

srcmsg

► C

Purpose

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

Syntax

►► -q nosrcmsg srcmsg ►►

See also “#pragma options” on page 325.

Notes

The compiler reconstructs the source line or partial source line to which the diagnostic message refers and displays it before the diagnostic message. A pointer to the column position of the error may also be displayed. Specifying **-qnosrcmsg** suppresses the generation of both the source line and the finger line, and the error message simply shows the file, line and column where the error occurred.

The reconstructed source line represents the line as it appears after macro expansion. At times, the line may be only partially reconstructed. The characters “...” at the start or end of the displayed line indicate that some of the source line has not been displayed.

The default (**-qnosrcmsg**) displays concise messages that can be parsed. Instead of giving the source line and pointers for each error, a single line is displayed, showing the name of the source file with the error, the line and character column position of the error, and the message itself.

Example

To compile `myprogram.c` so that the source line is displayed along with the diagnostic message when an error occurs, enter:

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

Related References

“Compiler Command Line Options” on page 61

“#pragma options” on page 325

staticinline

C++

Purpose

This option controls whether inline functions are treated as static or extern. By default, VisualAge C++ treats inline functions as extern.

Syntax

► — -q —  — ►

Example

Using the **-qstaticinline** option causes function `f` in the following declaration to be treated as static, even though it is not explicitly declared as such.

```
inline void f() { /*...*/};
```

Using the default, **-qnostaticinline**, gives `f` external linkage.

Related References

“Compiler Command Line Options” on page 61

statsym

► C ► C++

Purpose

Adds user-defined, nonexternal names that have a persistent storage class, such as initialized and uninitialized static variables, to the name list (the symbol table of `xcoff` objects).

Syntax

► — -q — nostatsym statsym — ◀◀

Default

The default is to not add static variables to the symbol table. However, static functions are added to the symbol table.

Example

To compile `myprogram.c` so that static symbols are added to the symbol table, enter:

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

Related References

“Compiler Command Line Options” on page 61

stdinc

> C > C++

Purpose

Specifies which directories are used for files included by the **#include** *<file_name>* and **#include** "file_name" directives. The **-qnostdinc** option excludes the standard include directories (**/usr/include** for C and **/usr/vacpp/include**, **/usr/include** for C++) from the search.

Syntax



See also "#pragma options" on page 325.

Notes

If you specify **-qnostdinc**, the compiler will not search the directory **/usr/include** for C files, or the directories **/usr/vacpp/include** and **/usr/include** for C++ files, unless you explicitly add them with the **-I***directory* option.

If a full (absolute) path name is specified, this option has no effect on that path name. It will still have an effect on all relative path names.

-qnostdinc is independent of **-qidirfirst**. (**-qidirfirst** searches the directory specified with **-I***directory* before searching the directory where the current source file resides.

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

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

Example

To compile `myprogram.c` so that the directory **/tmp/myfiles** is searched for a file included in `myprogram.c` with the **#include** "myinc.h" directive, enter:

```
xlc myprogram.c -qnostdinc -I/tmp/myfiles
```

Related References

"Compiler Command Line Options" on page 61

"I" on page 150

"idirfirst" on page 151

"#pragma options" on page 325

strict

C C++

Purpose

Turns off the aggressive optimizations that have the potential to alter the semantics of your program.

Syntax

►► -q{nostrict|strict} ◀◀

See also “#pragma options” on page 325.

Default

- **-qnostrict** with optimization levels of 3 or higher.
- **-qstrict** otherwise.

Notes

-qstrict turns off the following optimizations:

- Performing code motion and scheduling on computations such as loads and floating-point computations that may trigger an exception.
- Relaxing conformance to IEEE rules.
- Reassociating floating-point expressions.

This option is only valid with **-O2** or higher optimization levels.

-qstrict sets **-qfloat=noftint:nosqrt**.

-qnostrict sets **-qfloat=ftint:sqrt**.

You can use **-qfloat=ftint** and **-qfloat=rsqrt** to override the **-qstrict** settings.

For example:

- Using **-O3 -qstrict -qfloat=ftint** means that **-qfloat=ftint** is in effect, but there are no other aggressive optimizations.
- Using **-O3 -qnostrict -qfloat=norsqrt** means that the compiler performs all aggressive optimizations except **-qfloat=rsqrt**.

If there is a conflict between the options set with **-qnostrict** and **-qfloat=options**, the last option specified is recognized.

Example

To compile `myprogram.c` so that the aggressive optimizations of **-O3** are turned off, range checking is turned off **-qfloat=ftint**, and division by the result of a square root is replaced by multiplying by the reciprocal **-qfloat=rsqrt**, enter:

```
xlc myprogram.c -O3 -qstrict -qfloat=ftint:rsqrt
```

Related References

“Compiler Command Line Options” on page 61

“float” on page 131

“O, optimize” on page 215

“#pragma options” on page 325

strict_induction

► C ► C++

Purpose

Disables loop induction variable optimizations that have the potential to alter the semantics of your program. Such optimizations can change the result of a program if truncation or sign extension of a loop induction variable should occur as a result of variable overflow or wrap-around.

Syntax

►► -q nostrict_induction strict_induction ◄◄

Default

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

Notes

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

Related References

“Compiler Command Line Options” on page 61

“O, optimize” on page 215

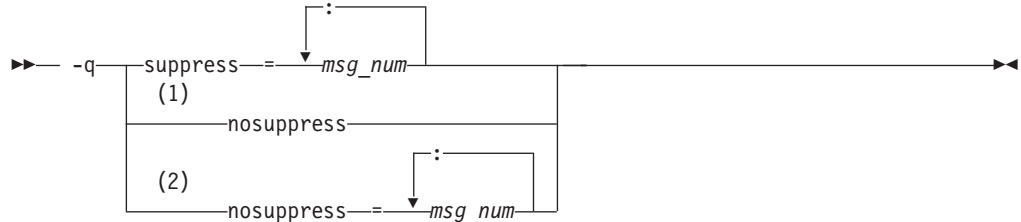
suppress

C C++

Purpose

Prevents the specified compiler or driver informational or warning messages from being displayed or added to the listings.

Syntax



Notes:

- 1 C only
- 2 C++ only

Notes

This option suppresses compiler messages only, and has no effect on linker or operating system messages.

To suppress IPA messages, enter **-qsuppress** before **-qipa** on the command line.

Compiler messages that cause compilation to stop, such as (S) and (U) level messages, or other messages depending on the setting of the **-qhalt** compiler option, cannot be suppressed. For example, if the **-qhalt=w** compiler option is set, warning messages will not be suppressed by the **-qsuppress** compiler option.

The **-qnosuppress** compiler option cancels previous settings of **-qsuppress**.

Example

If your program normally results in the following output:

```
"t.c", line 1.1:1506-224 (I) Incorrect #pragma ignored
```

you can suppress the message by compiling with:

```
x1C myprogram.c -qsuppress
```

Related References

"Compiler Command Line Options" on page 61

"halt" on page 143

"ipa" on page 163

symtab

C C++

Purpose

Controls the symbol table.

Syntax

►► — -q-symtab = — [unref] —————►►
 └──static──┘

where:

unref Specifies that all **typedef** declarations, **struct**, **union**, and **enum** type definitions are included for processing by the Distributed Debugger.

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

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

Using **-qsymtab=unref** may make your object and executable files larger.

static Adds user-defined, nonexternal names that have a persistent storage class, such as initialized and uninitialized static variables, to the name list (the symbol table of **xcoff** objects).

The default is to not add static variables to the symbol table.

Examples

To compile `myprogram.c` so that static symbols are added to the symbol table, enter:

```
x1C myprogram.c -qsymtab=static
```

To include all symbols in `myprogram.c` in the symbols table for use with the Distributed Debugger, enter:

```
x1C myprogram.c -g -qsymtab=unref
```

Related References

“Compiler Command Line Options” on page 61

“g” on page 141

syntaxonly

► C

Purpose

Causes the compiler to perform syntax checking without generating an object file.

Syntax

►► — -q—syntaxonly ————— ◀◀

Notes

The **-P**, **-E**, and **-C** options override the **-qsyntaxonly** option, which in turn overrides the **-c** and **-o** options.

The **-qsyntaxonly** option suppresses only the generation of an object file. All other files (listings, etc) are still produced if their corresponding options are set.

Examples

To check the syntax of `myprogram.c` without generating an object file, enter:

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

or

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

Note that in the second example, the **-qsyntaxonly** option overrides the **-o** option so no object file is produced.

Related References

“Compiler Command Line Options” on page 61

“C” on page 93

“c” on page 94

“E” on page 115

“o” on page 219

“P” on page 222

t

> C > C++

Purpose

Adds the prefix specified by the **-B** option to the designated programs.

Syntax



where programs are:

Program	Description
c	Compiler front end
b	Compiler back end
p	Compiler preprocessor
a	Assembler
I	Interprocedural Analysis tool - compile phase
L	Interprocedural Analysis tool - link phase
l	Linkage editor
E	CreateExportList utility
m	Linkage helper (munch utility)

Notes

This option must be used together with the **-B** option.

Default

If **-B** is specified but *prefix* is not, the default prefix is **/lib/o**. If **-Bprefix** is not specified at all, the prefix of the standard program names is **/lib/n**.

If **-B** is specified but **-tprograms** is not, the default is to construct path names for all the standard program names: (**c**, **b**, **I**, **a**, **l**, and **m**).

Example

To compile myprogram.c so that the name **/u/newones/compilers/** is prefixed to the compiler and assembler program names, enter:

```
xlc myprogram.c -B/u/newones/compilers/ -tca
```

Related References

"Compiler Command Line Options" on page 61

"B" on page 88

tabsize

► C ► C++

Purpose

Changes the length of tabs as perceived by the compiler.

Syntax

►► — -q—tabsize—=—*n*—————▶▶

where *n* is the number of character spaces representing a tab in your source program.

Notes

This option only affects error messages that specify the column number at which an error occurred. For example, the compiler will consider tabs as having a width of one character if you specify **-qtabsize=1**. In this case, you can consider one character position (where each character and each tab equals one position, regardless of tab length) as being equivalent to one character column.

Related References

“Compiler Command Line Options” on page 61

tbtable

C C++

Purpose

Generates a traceback table that contains information about each function, including the type of function as well as stack frame and register information. The traceback table is placed in the text segment at the end of its code.

Syntax

```
-q-tbtable=  
├── none  
├── full  
└── small
```

where suboptions are::

none	No traceback table is generated. The stack frame cannot be unwound so exception handling is disabled.
full	A full traceback table is generated, complete with name and parameter information. This is the default if -qnoopt or -g are specified.
small	The traceback table generated has no name or parameter information, but otherwise has full traceback capability. This is the default if you have specified optimization and have not specified -g .

See also “#pragma options” on page 325.

Notes

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

Many performance measurement tools require a full traceback table to properly analyze optimized code. The compiler configuration file contains entries to accommodate this requirement. If you do not require full traceback tables for your optimized code, you can save file space by making the following changes to your compiler configuration file:

1. Remove the **-qtbtable=full** option from the **options** lines of the C or C++ compilation stanzas.
2. Remove the **-qtbtable=full** option from the **xlCopt** line of the **DFLT** stanza.

With these changes, the defaults for the **tbtable** option are:

- When compiling with optimization options set, **-qtbtable=small**
- When compiling with no optimization options set, **-qtbtable=full**

See *Interlanguage Calls - Traceback Table* for a brief description of traceback tables.

Related References

“Compiler Command Line Options” on page 61

“g” on page 141

“O, optimize” on page 215

“#pragma options” on page 325

“Interlanguage Calls - Traceback Table” on page 56

See also:

ld command in *Commands Reference, Volume 5: s through u*

tempinc

C++

Purpose

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

Syntax



Default

The default is to generate the separate include files and place them in the **tempinc** directory of the current directory at compile time.

The **-qtempinc** and **-qtemplateregistry** compiler options are mutually exclusive. Specifying **-qtempinc** implies **-qnotemplateregistry**. However, specifying **-qnotempinc** does not imply **-qtemplateregistry**.

Notes

When you specify **-qtempinc**, the compiler assigns a value of 1 to the `__TEMPINC__` macro. This assignment will not occur if **-qnotempinc** has been specified.

Example

To compile the file `myprogram.c` and place the generated include files for the template functions in the **/tmp/mytemplates** directory, enter:

```
x1C myprogram.C -qtempinc=/tmp/mytemplates
```

Related Tasks

“Structure a Program that Uses Templates” on page 35

“Use `-qtempinc` to Generate Template Functions Automatically” on page 38

“Use `-qnotempinc` to Define Template Functions” on page 42

Related References

“Compiler Command Line Options” on page 61

“templateregistry” on page 271

templaterecompile

C++

Purpose

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

Syntax

►► -q templaterestry notemplaterestry ►►

Notes

The **-qtemplaterestry** option helps to manage dependencies between compilation units that have been compiled using the **-qtemplaterestry** option. Given a program in which multiple compilation units reference the same template instantiation, the **-qtemplaterestry** option nominates a single compilation unit to contain the instantiation. No other compilation units will contain this instantiation. Duplication of object code is thereby avoided. If a source file that has been compiled previously is compiled again, the **-qtemplaterestry** option consults the template registry to determine whether changes to this source file require the recompile of other compilation units. This can occur when the source file has changed in such a way that it no longer references a given instantiation and the corresponding object file previously contained the instantiation. If so, affected compilation units will be recompiled automatically.

The **-qtemplaterestry** option requires that object files generated by the compiler remain in the subdirectory to which they were originally written. If your automated build process moves object files from their original subdirectory, use the **-qnotemplaterestry** option whenever **-qtemplaterestry** is enabled.

Related Tasks

“Structure a Program that Uses Templates” on page 35

“Use -qtemplaterestry to Define Template Functions” on page 43

Related References

“Compiler Command Line Options” on page 61

“templaterestry” on page 271

“tempinc” on page 269

tempmax

C++

Purpose

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

Syntax

►► `-qtempmax=` `number` ◀◀

Notes

Specify the maximum number of template files by giving *number* a value between 1 and 99999.

Instantiations are spread among the template include files.

This option should be used when the size of files generated by the `-qtempinc` option become very large and take a significant amount of time to recompile when a new instance is created.

Related Tasks

“Structure a Program that Uses Templates” on page 35

Related References

“Compiler Command Line Options” on page 61

“tempinc” on page 269

threaded

► C ► C++

Purpose

Indicates to the compiler that the program will run in a multi-threaded environment. Always use this option when compiling or linking multi-threaded applications. This option ensures that all optimizations are thread-safe.

Syntax

►► -q {nothreaded | threaded} ◀◀

Default

The default is **-qthreaded** when compiling with **_r** invocation modes, and **-qnothreaded** when compiling with other invocation modes.

Notes

This option applies to both compile and linkage editor operations.

To maintain thread safety, a file compiled with the **-qthreaded** option, whether explicitly by option selection or implicitly by choice of **_r** compiler invocation mode, must also be linked with the **-qthreaded** option.

This option does not make code thread-safe, but it will ensure that code already thread-safe will remain so after compile and linking.

Related References

“Compiler Command Line Options” on page 61

“smp” on page 252

tmp1parse

C++

Purpose

This option controls whether parsing and semantic checking are applied to template definition (class template definitions, function bodies, member function bodies, and static data member initializers) or only to template instantiations. VisualAge C++ can check function bodies and variable initializers in template definitions and produce error or warning messages.

Syntax



where suboptions are:

no	Do not parse the template definitions. This reduces the number of errors issued in code written for previous versions of VisualAge C++ and predecessor products. This is the default.
warn	Parses template definitions and issues warning messages for semantic errors.
error	Treats problems in template definitions as errors, even if the template is not instantiated.

Notes

This option applies to template definitions, not their instantiations. Regardless of the setting of this option, error messages are produced for problems that appear outside definitions. For example, errors found during the parsing or semantic checking of constructs such as the following, always cause error messages:

- return type of a function template
- parameter list of a function template

Related Tasks

“Structure a Program that Uses Templates” on page 35

Related References

“Compiler Command Line Options” on page 61

tocdata

► C ► C++

Purpose

Marks data as local.

Syntax

►► -q notocdata tocdata ►►

Notes

Local variables are statically bound with the functions that use them. **-qtocdata** changes the default to assume that all variables are local. **-qtocdata** marks the named variables as local. The default is not changed. Performance may decrease if an imported variable is assumed to be local.

Imported variables are dynamically bound with a shared portion of a library. **-qnotocdata** changes the default to assume that all variables are imported. The default is not changed.

Conflicts among the data-marking options are resolved in the following manner:

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

Related References

“Compiler Command Line Options” on page 61

tocmerge

> C > C++

Purpose

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

Syntax

►► — -q — notocmerge
tocmerge —————►►

Notes

This compiler option enables TOC merging to reduce TOC pointer loads and improves the scheduling of external loads. If **-qtocmerge** is specified, the compiler reads from the file specified in the **-bImportfile** linker option. If **-qtocmerge** is specified but no import filename is specified, the option is ignored and a warning message is issued.

Related References

“Compiler Command Line Options” on page 61

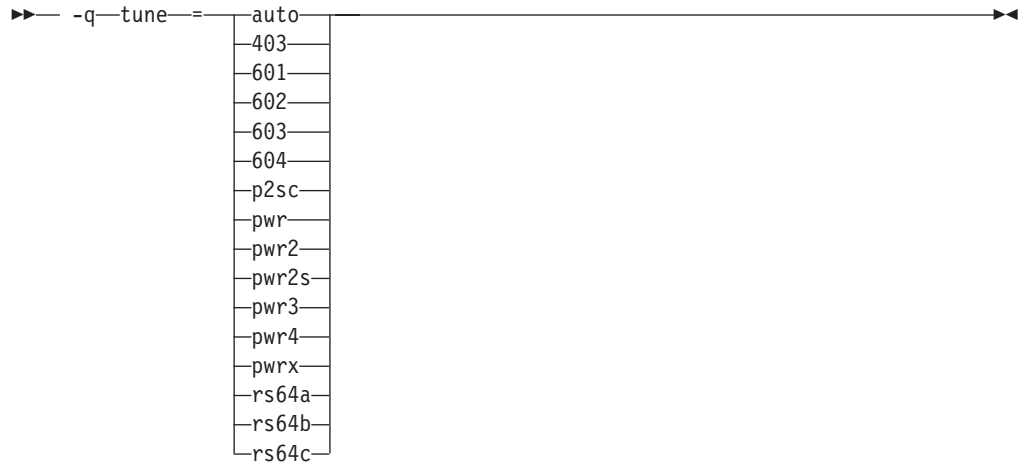
tune

C C++

Purpose

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

Syntax



where architecture suboptions are:

Suboption Description

auto	Produces object code optimized for the hardware platform on which it is compiled.
403	Produces object code optimized for the PowerPC 403 processor.
601	Produces object code optimized for the PowerPC 601 processor.
602	Produces object code optimized for the PowerPC 602 processor.
603	Produces object code optimized for the PowerPC 603 processor.
604	Produces object code optimized for the PowerPC 604 processor.
p2sc	Produces object code optimized for the PowerPC P2SC processor.
pwr	Produces object code optimized for the POWER hardware platforms.
pwr2	Produces object code optimized for the POWER2 hardware platforms.
pwr2s	Produces object code optimized for the POWER2 hardware platforms, avoiding certain quadruple-precision instructions that would slow program performance.
pwr3	Produces object code optimized for the POWER3 hardware platforms.
pwr4	Produces object code optimized for the POWER4 hardware platforms.
pwrx	Produces object code optimized for the POWER2 hardware platforms (same as -qtune=pwr2).
rs64a	Produces object code optimized for the RS64A processor.
rs64b	Produces object code optimized for the RS64B processor.
rs64c	Produces object code optimized for the RS64C processor.

See also “#pragma options” on page 325.

Default

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

- If **-qtune** is specified without **-qarch**, the compiler uses **-qarch=com**.

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

```
TUNE=DEFAULT
```

To find the actual default **-qtune** setting for a given **-qarch** setting, refer to the table in “Acceptable Compiler Mode and Processor Architecture Combinations” on page 373.

Notes

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

- **-qarch=***suboption* specifies the architecture for which the instructions are to be generated, and,
- **-qtune=***suboption* specifies the target platform for which the code is optimized.

Example

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

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

Related Tasks

“Specify Compiler Options for Architecture-Specific, 32- or 64-bit Compilation” on page 29

Related References

“Compiler Command Line Options” on page 61

“arch” on page 83

“Acceptable Compiler Mode and Processor Architecture Combinations” on page 373

twolink

C++

Purpose

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

Syntax

►► -q notwmlink
twolink ◀◀

Notes

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

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

Default

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

Example

Given the include file foo.h:

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

and the C++ program t.C:

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

and the program t2.C:

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

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

```
x1C -c t.C t2.C
```

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

```
x1C t.o t2.o
```

Invoking a.out produces:

```
in foo
in ~foo
```

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

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

and then use the default compiler command:

```
x1C t2.o t.a
```

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

```
in foo  
in ~foo
```

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

```
x1C -qtwolink t2.o t.a
```

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

Related References

“Compiler Command Line Options” on page 61

“funcsect” on page 139

“ttable” on page 268

Also, on the Web see:

ar Command section in Commands Reference, Volume 1: a through c for information about profiling.

U

► C ► C++

Purpose

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

Syntax

►► -U—*name*◀◀

Notes

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

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

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

Example

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

```
x1C myprogram.c -UCOUNT
```

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

Related References

“Compiler Command Line Options” on page 61

“D” on page 106

unique

C++

Purpose

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

Syntax

► — -q — nunique — unique — ►

Notes

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

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

Example

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

```
sqlpreprocess file1.sql > t.C
x1C -qunique t.C -o file1.o
rm -f t.C
sqlpreprocess file2.sql > t.C
x1C -qunique t.C -o file2.o
rm -f t.C
x1C file1.o file2.o
```

Following is a sample **make** file for the above example:

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

Related References

“Compiler Command Line Options” on page 61

unroll

C C++

Purpose

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

Syntax



where:

- qunroll=auto Leaves the decision to unroll loops to the compiler.
- qunroll or -qunroll=yes Is a suggestion to the compiler to unroll loops.
- qnounroll or -qunroll=no Instructs the compiler to not unroll loops.

See also “#pragma unroll” on page 342 and “#pragma options” on page 325.

Notes

Specifying **-qunroll** is equivalent to specifying **-qunroll=yes**.

When **-qunroll**, **-qunroll=yes**, or **-qunroll=auto** is specified, the bodies of inner loops will be unrolled, or duplicated, by the optimizer. The optimizer determines and applies the best unrolling factor for each loop. In some cases, the loop control may be modified to avoid unnecessary branching.

To see if the **unroll** option improves performance of a particular application, you should first compile the program with usual options, then run it with a representative workload. You should then recompile with command line **-qunroll** option and/or the **unroll** pragmas enabled, then rerun the program under the same conditions to see if performance improves.

You can use the **#pragma unroll** directive to gain more control over unrolling. Setting this pragma overrides the **-qunroll** compiler option setting.

Examples

1. In the following examples, unrolling is disabled:

```
x1C -qnounroll file.c
```

```
x1C -qunroll=no file.c
```

2. In the following examples, unrolling is enabled:

```
x1C -qunroll file.c
```

```
x1C -qunroll=yes file.c
```

```
x1C -qunroll=auto file.c
```

3. See “#pragma unroll” on page 342 for examples of how program code is unrolled by the compiler.

Related References

"Compiler Command Line Options" on page 61

"#pragma options" on page 325

"#pragma unroll" on page 342

unwind

► C ► C++

Purpose

Informs the compiler that the application does not rely on any program stack unwinding mechanism.

Syntax

►► -q unwind nounwind ◀◀

Notes

Selecting the **-qnounwind** option can improve optimization of non-volatile register saves and restores.

► C++ For C++ programs, specifying **-qnounwind** will also imply **-qnoeh**.

Related References

“Compiler Command Line Options” on page 61

“eh” on page 118

upconv

► C

Purpose

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

Syntax

► -q noupconv upconv ►

See also “#pragma options” on page 325.

Notes

The **-qupconv** option promotes any **unsigned** type smaller than an **int** to an **unsigned int** instead of to an **int**.

Unsignedness preservation is provided for compatibility with older dialects of C. The ANSI C standard requires value preservation as opposed to unsignedness preservation.

Default

The default is **-qnoupconv**, except when **-qlanglvl=ext**, in which case the default is **-qupconv**. The compiler does not preserve the **unsigned** specification.

The default compiler action is for integral promotions to convert a **char**, **short int**, **int bitfield** or their **signed** or **unsigned** types, or an **enumeration** type to an **int**. Otherwise, the type is converted to an **unsigned int**.

Example

To compile myprogram.c so that all **unsigned** types smaller than **int** are converted to **unsigned int**, enter:

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

The following short listing demonstrates the effect of **-qupconv**:

```
#include <stdio.h>
int main(void) {
    unsigned char zero = 0;
    if (-1 < zero)
        printf("Value-preserving rules in effect\n");
    else
        printf("Unsignedness-preserving rules in effect\n");
    return 0;
}
```

Related References

“Compiler Command Line Options” on page 61

“langlvl” on page 175

V

► C ► C++

Purpose

Instructs the compiler to report information on the progress of the compilation, names the programs being invoked within the compiler and the options being specified to each program. Information is displayed in a format similar to that of shell commands.

Syntax

►► -V ◀◀

Notes

The `-V` option is overridden by the `-#` option.

Example

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

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

Related References

“Compiler Command Line Options” on page 61

V

> C > C++

Purpose

Instructs the compiler to report information on the progress of the compilation, names the programs being invoked within the compiler and the options being specified to each program. Information is displayed to standard output.

Syntax

▶▶ -v ◀◀

Notes

The `-v` option is overridden by the `-#` option.

Example

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

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

Related References

“Compiler Command Line Options” on page 61

vftable

► C++

Purpose

Controls the generation of virtual function tables.

Syntax

► — -q — novftable —
 └─ vftable ─┘

Default

The default is to define the virtual function table for a class if the current compilation unit contains the body of the first non-inline virtual member function declared in the class member list.

Notes

Specifying **-qvftable** generates virtual function tables for all classes with virtual functions that are defined in the current compilation unit.

If you specify **-qnovftable**, no virtual function tables are generated in the current compilation unit.

Example

To compile the file `myprogram.c` so that no virtual function tables are generated, enter:

```
x1C myprogram.C -qnovftable
```

Related References

“Compiler Command Line Options” on page 61

W

> C > C++

Purpose

Passes the listed options to a designated compiler program.

Syntax



where programs are:

program	Description
a	Assembler
b	Compiler back end
c	Compiler front end
I	Interprocedural Analysis tool
l	linkage editor
p	compiler preprocessor

Notes

When used in the configuration file, the **-W** option accepts the escape sequence backslash comma (\,) to represent a comma in the parameter string.

Example

To compile myprogram.c so that the *option* **-pg** is passed to the linkage editor (**I**) and the assembler (**a**), enter:

```
x1C myprogram.c -Wl,-pg -Wa,-pg
```

In a configuration file, use the \, sequence to represent the comma (,).

```
-Wl\,-pg,-Wa\,-pg
```

Related References

“Compiler Command Line Options” on page 61

W

► C ► C++

Purpose

Requests that warnings and lower-level messages be suppressed. Specifying this option is equivalent to specifying **-qflag=e:e**.

Syntax

►► -w ◀◀

Example

To compile myprogram.c so that no warning messages are displayed, enter:

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

Related References

“Compiler Command Line Options” on page 61

“flag” on page 130

warn64

> C > C++

Purpose

Enables checking for possible *long-to-integer* truncation.

Syntax

▶▶ — -q—warn64 —————▶▶

Notes

All generated messages have level Informational.

This option functions in either 32- or 64-bit compiler modes. In 32-bit mode, it functions as a preview aid to discover possible 32- to 64-bit migration problems.

Informational messages are displayed where data conversion may cause problems in 64-bit compilation mode, such as:

- truncation due to explicit or implicit conversion of **long** types into **int** types
- unexpected results due to explicit or implicit conversion of **int** types into **long** types
- invalid memory references due to explicit conversion by cast operations of **pointer** types into **into** types
- invalid memory references due to explicit conversion by cast operations of **int** types into **pointer** types
- problems due to explicit or implicit conversion of **constants** into **long** types
- problems due to explicit or implicit conversion by cast operations of **constants** into **pointer** types
- conflicts with pragma options **arch** in source files and on the command line

Related References

“Compiler Command Line Options” on page 61

“32, 64” on page 73

xcall

► C ► C++

Purpose

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

Syntax

►► -q  

Notes

-qxcall generates slower code than -qnoxcall.

Example

To compile myprogram.c so all static routines are compiled as external routines, enter:

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

Related References

“Compiler Command Line Options” on page 61

xref

C C++

Purpose

Produces a compiler listing that includes a cross-reference listing of all identifiers.

Syntax

`-q` `noxref` `xref`

where:

<code>xref=full</code>	Reports all identifiers in the program.
<code>xref</code>	Reports only those identifiers that are used.

See also “#pragma options” on page 325.

Notes

The `-qnoprint` option overrides this option.

Any function defined with the `#pragma mc_func function_name` directive is listed as being defined on the line of the `#pragma` directive.

Example

To compile `myprogram.c` and produce a cross-reference listing of all identifiers whether they are used or not, enter:

```
xlc myprogram.c -qxref=full -qattr
```

A typical cross-reference listing has the form:

Identifier name	Description of the item
<code>xy</code>	<code>auto int in function adder</code>
	<code>0-59Y 0-36.12Z 0-48.12Z</code>
	Function invocation
	Column number
	Line number
	File
	Function definition

Related References

“Compiler Command Line Options” on page 61

“print” on page 231

“#pragma mc_func” on page 321

“#pragma options” on page 325

y

► C ► C++

Purpose

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

Syntax



where suboptions are:

n	Round to the nearest representable number. This is the default.
m	Round toward minus infinity.
p	Round toward plus infinity.
z	Round toward zero.

Example

To compile myprogram.c so that constant floating-point expressions are rounded toward zero at compile time, enter:

```
x1C myprogram.c -yz
```

Related References

“Compiler Command Line Options” on page 61

Z

► C ► C++

Purpose

This option specifies a prefix for the library search path.

Syntax

►► — *-Z—string* —————►►

Notes

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

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

Related References

“Compiler Command Line Options” on page 61

General Purpose Pragmas

The pragmas listed below are available for general programming use. Unless noted otherwise, pragmas can be used in both C and C++ programs.

Language Application		#pragma	Description
> C	> C++	#pragma align	Aligns data items within structures.
> C		#pragma alloca	Provides an inline version of the function alloca(size_t size) .
> C	> C++	#pragma chars	Sets the sign type of character data.
> C	> C++	#pragma comment	Places a comment into the object file.
	> C++	#pragma define	Forces the definition of a template class without actually defining an object of the class.
> C	> C++	#pragma disjoint	Lists the identifiers that are not aliased to each other within the scope of their use.
> C	> C++	#pragma enum	Specifies the size of enum variables that follow.
> C	> C++	#pragma execution_frequency	Marks program source code that is not frequently executed.
	> C++	#pragma hashome	Informs the compiler that the specified class has a home module that will be specified by the IsHome pragma.
> C	> C++	#pragma ibm snapshot	Sets a debugging breakpoint at the point of the pragma, and defines a list of variables to examine when program execution reaches that point.
	> C++	#pragma implementation	Tells the compiler the name of the file containing the function-template definitions that correspond to the template declarations in the include file which contains the pragma.
> C	> C++	#pragma info	Controls the diagnostic messages generated by the info(...) compiler options.
	> C++	#pragma ishome	Informs the compiler that the specified class's home module is the current compilation unit.
> C	> C++	#pragma isolated_call	Lists functions that do not alter data objects visible at the time of the function call.
> C		#pragma langlvl	Selects the C or C++ language level for compilation.
> C	> C++	#pragma leaves	Takes a function name and specifies that the function never returns to the instruction after the function call.
> C	> C++	#pragma map	Tells the compiler that all references to an identifier are to be converted to a new name.
> C	> C++	#pragma mc_func	Specifies machine instructions for a particular function.
	> C++	#pragma namemangling	Sets the name mangling scheme and maximum length of external names generated from source code.

Language Application	#pragma	Description
> C++	#pragma nameManglingRule	Instructs the compiler whether or not to mangle function names according to their function parameter types.
> C++	#pragma object_model	Specifies the object model to use for the structures, unions, and classes that follow it.
> C > C++	#pragma options	Specifies options to the compiler in your source program.
> C > C++	#pragma option_override	Specifies alternate optimization options for specific functions.
> C > C++	#pragma pack	Modifies the current alignment rule for members of structures that follow this pragma.
> C++	#pragma pass_by_value	Specifies how classes containing const or reference members are passed in function arguments. All classes in the compilation unit are affected by this option.
> C++	#pragma priority	Specifies the order in which static objects are to be initialized at run time.
> C > C++	#pragma reachable	Declares that the point after the call to a routine marked reachable can be the target of a branch from some unknown location.
> C > C++	#pragma reg_killed_by	Specifies those registers which value will be corrupted by the specified function. It must be used together with #pragma mc_func.
> C++	#pragma report	Controls the generation of specific messages.
> C > C++	#pragma strings	Sets storage type for strings.
> C > C++	#pragma unroll	Unrolls inner loops in the program, This can help improve program performance.

Related Concepts

“Program Parallelization” on page 9

Related Tasks

“Specify Compiler Options in Your Program Source Files” on page 27

“Control Parallel Processing with Pragmas” on page 45

Related References

“Pragmas to Control Parallel Processing” on page 344

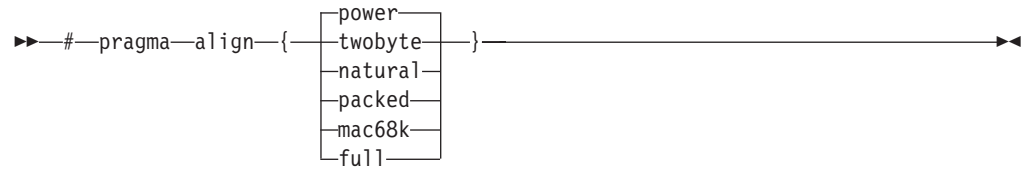
#pragma align

C C++

Description

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

Syntax



Related References

"General Purpose Pragmas" on page 297

"align" on page 77

#pragma alloca

► C

Description

The `#pragma alloca` directive specifies that the compiler should provide an inline version of the function `alloca(size_t <size>)`. The function `alloca(size_t <size>)` can be used to allocate space for an object. The amount of space allocated is determined by the value of `<size>`, which is measured in bytes. The allocated space is put on the stack.

Syntax

► `#pragma alloca` ◀

Notes

You must specify the `#pragma alloca` directive or `-ma` compiler option to have the compiler provide an inline version of `alloca`.

Once specified, it applies to the rest of the file and cannot be turned off. If a source file contains any functions that you want compiled without `#pragma alloca`, place these functions in a different file.

Related References

“General Purpose Pragmas” on page 297

“`alloca`” on page 81

#pragma chars

► C ► C++

Description

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

Syntax

```
► #pragma chars { unsigned | signed } ◀
```

Notes

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

Once specified, the pragma applies to the entire file and cannot be turned off. If a source file contains any functions that you want to be compiled without **#pragma chars**, place these functions in a different file. If the pragma is specified more than once in the source file, the first one will take precedence.

Note: The default character type behaves like an unsigned char.

Related References

"General Purpose Pragas" on page 297

"chars" on page 97

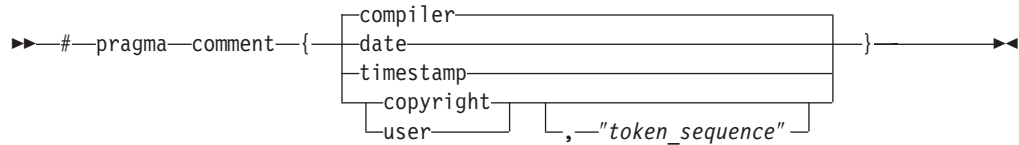
#pragma comment

> C > C++

Description

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

Syntax



where:

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

Related References

“General Purpose Pragmas” on page 297

#pragma define

C++

Description

The **#pragma define** directive forces the definition of a template class without actually defining an object of the class. This pragma is only provided for backward compatibility purposes.

Syntax

```
▶▶ #pragma define (—template_classname—) ▶▶
```

where the *template_classname* is the name of the template to be defined.

Notes

A user can explicitly instantiate a class, function or member template specialization by using a construct of the form:

```
template declaration
```

For example:

```
#pragma define(Array<char>)
```

is equivalent to:

```
template class Array<char>;
```

This pragma must be defined in global scope (i.e. it cannot be enclosed inside a function/class body). It is used when organizing your program for the efficient or automatic generation of template functions.

Related References

“General Purpose Pragas” on page 297

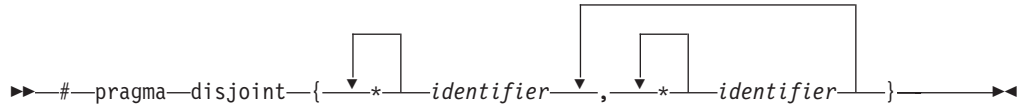
#pragma disjoint

C C++

Description

The **#pragma disjoint** directive lists the identifiers that are not aliased to each other within the scope of their use.

Syntax



Notes

The directive informs the compiler that none of the identifiers listed shares the same physical storage, which provides more opportunity for optimizations. If any identifiers actually share physical storage, the pragma may cause the program to give incorrect results.

An identifier in the directive must be visible at the point in the program where the pragma appears. The identifiers in the disjoint name list cannot refer to any of the following:

- a member of a structure, or union
- a structure, union, or enumeration tag
- an enumeration constant
- a typedef name
- a label

This pragma can be disabled with the **-qignprag** compiler option.

Example

```
int a, b, *ptr_a, *ptr_b;
#pragma disjoint(*ptr_a, b) // *ptr_a never points to b
#pragma disjoint(*ptr_b, a) // *ptr_b never points to a
one_function()
{
    b = 6;
    *ptr_a = 7; // Assignment does not alter the value of b
    another_function(b); // Argument "b" has the value 6
}
```

Because external pointer *ptr_a* does not share storage with and never points to the external variable *b*, the assignment of 7 to the object that *ptr_a* points to will not change the value of *b*. Likewise, external pointer *ptr_b* does not share storage with and never points to the external variable *a*. The compiler can assume that the argument of *another_function* has the value 6 and will not reload the variable from memory.

Related References

“General Purpose Pragmas” on page 297

“ignprag” on page 153

“alias” on page 75

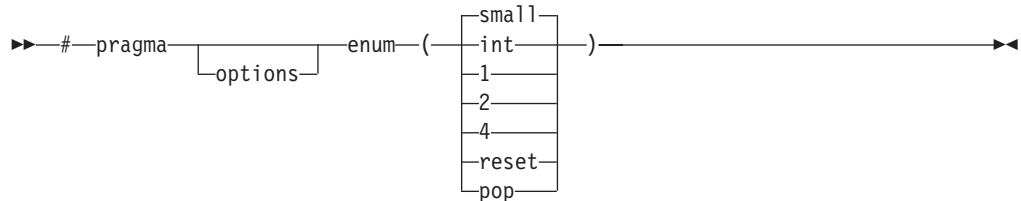
#pragma enum

> C > C++

Description

The **#pragma enum** directive specifies the size of enum variables that follow. The size at the left brace of a declaration is the one that affects that declaration, regardless of whether further enum directives occur within the declaration. This pragma pushes a value on a stack each time it is used, with a reset option available to return to the previously pushed value.

Syntax



where *option* can be substituted with one of the following:

small	enum size is the smallest integral type that can contain all variables.
int	enum size is 4
1	enum size is 1
2	enum size is 2
4	enum size is 4
pop	the option will reset the enum size to the one before the previously set enum size.
reset	the option is an alternative method of resetting the enum size to the one before the previously set enum size. This option is provided for backwards compatibility.

Notes

Popping on an empty stack generates a warning message and the enum value remains unchanged.

The **#pragma enum** directive overrides the **-qenum** compiler option.

Example

```
#pragma enum(1)  
#pragma enum(2)  
#pragma enum(4)  
#pragma enum(pop) /* will reset enum size to 2 */  
#pragma enum(reset) /* will reset enum size to 1 */  
#pragma enum(pop) /* will reset enum size to default
```

Related References

“General Purpose Pragmas” on page 297

“enum” on page 119

#pragma execution_frequency

C C++

Description

The `#pragma execution_frequency` directive lets you mark program source code that is not frequently executed.

Syntax

`#pragma execution_frequency(—very_low—)`

Notes

Use this pragma to mark program source code that will be executed only infrequently. The pragma must be placed within block scope, and acts on the closest point of branching.

The pragma is used as a hint to the optimizer. If optimization is not selected, this pragma has no effect.

Examples

1. This pragma is used in an `if` statement block to mark code that is executed infrequently.

```
int *array = (int *) malloc(10000);

if (array == NULL) {
    /* Block A */
    #pragma execution_frequency(very_low)
    error();
}
```

The code block "Block B" would be marked as infrequently executed and "Block C" is likely to be chosen during branching.

```
if (Foo > 0) {
    #pragma execution_frequency(very_low)
    /* Block B */
    doSomething();
} else {
    /* Block C */
    doAnotherThing();
}
```

2. This pragma is used in a `switch` statement block to mark code that is executed infrequently.

```
while (counter > 0) {
    #pragma execution_frequency(very_low)
    doSomething();
} /* This loop is unlikely to be executed. Even if it is executed,
   it is likely that only very few iterations are executed. */

switch (a) {
    case 1:
        doOneThing();
        break;
    case 2:
        #pragma execution_frequency(very_low)
        doTwoThings();
        break;
    default:
        doNothing();
} /* The second case is not likely chosen. */
```

3. This pragma cannot be used at file scope. It can be placed anywhere within a block scope and it affects the closest branching.

```
int a;
#pragma execution_frequency(very_low)
int b;

int foo(boolean boo) {
    #pragma execution_frequency(very_low)
    char c;

    if (boo) {
        /* Block A */
        doSomething();
        {
            /* Block C */
            doSomethingAgain();
            #pragma execution_frequency(very_low)
            doAnotherThing();
        }
    } else {
        /* Block B */
        doNothing();
    }

    return 0;
}

#pragma execution_frequency(very_low)
```

The first and fourth pragmas are invalid, while the second and third are valid. However, only the third pragma has effect and it affects whether branching to Block A or Block B in the decision "if (**boo**)". The second pragma is ignored by the compiler.

Related References

"General Purpose Pragmas" on page 297

#pragma hashome

C++

Description

The **#pragma hashome** directive informs the compiler that the specified class has a home module that will be specified by **#pragma ishome**. This class's virtual function table, along with certain inline functions, will not be generated as static. Instead, they will be referenced as externals in the compilation unit of the class in which **#pragma ishome** was specified.

Syntax

▶▶ #pragma hashome (—*className* [AllInlines]) ▶▶

where:

className specifies the name of a class that requires the above mentioned external referencing. *className* must be a class and it must be defined.

AllInlines specifies that all inline functions from within *className* should be referenced as being external. This argument is case insensitive

Notes

A warning will be produced if there is a **#pragma ishome** without a matching **#pragma hashome**.

Related References

"General Purpose Pragmas" on page 297
"#pragma ishome" on page 314

#pragma ibm snapshot

► C ► C++

Description

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

Syntax

►► #pragma ibm snapshot (*variable_name*)

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

Notes

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

Example

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

Related References

“General Purpose Pragmas” on page 297

#pragma implementation

► C++

Description

The **#pragma implementation** directive tells the compiler the name of the template header file containing the function-template definitions. These definitions correspond to the template declarations in the include file containing the pragma.

Syntax

►► #pragma implementation (—*string_literal*—) ◀◀

Notes

This pragma can appear anywhere that a declaration is allowed. It is used when organizing your program for the efficient or automatic generation of template functions.

Related References

“General Purpose Pragmas” on page 297

“tempmax” on page 272

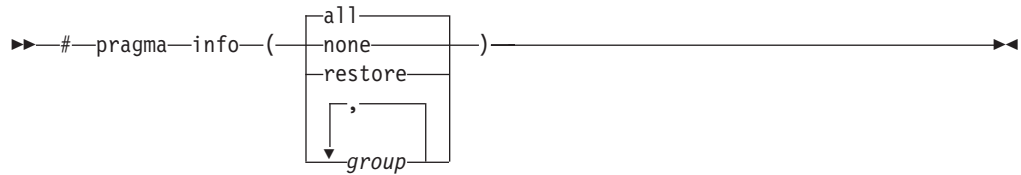
#pragma info

► C ► C++

Description

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

Syntax



where:

- `all` Turns on all diagnostic checking.
- `none` Turns off all diagnostic suboptions for specific portions of your program
- `restore` Restores the options that were in effect before the previous `#pragma info` directive.

group Generates or suppresses all messages associated with the specified diagnostic *group*. More than one *group* name in the following list can be specified.

<i>group</i>	Type of messages returned or suppressed
c99 noc99	C code that may behave differently between C89 and C99 language levels.
cls nocls	Classes
cmp nocmp	Possible redundancies in unsigned comparisons
cond nocnd	Possible redundancies or problems in conditional expressions
cns nocns	Operations involving constants
cnv nocnv	Conversions
dc1 nodc1	Consistency of declarations
eff noeff	Statements and pragmas with no effect
enu noenu	Consistency of enum variables
ext noext	Unused external definitions
gen nogen	General diagnostic messages
gnr nognr	Generation of temporary variables
got nogot	Use of goto statements
ini noini	Possible problems with initialization
inl noinl	Functions not inlined
lan nolan	Language level effects
obs noobs	Obsolete features
ord noord	Unspecified order of evaluation
par nopar	Unused parameters
por nopor	Nonportable language constructs
ppc noppc	Possible problems with using the preprocessor
ppt noppt	Trace of preprocessor actions
pro nopro	Missing function prototypes
rea norea	Code that cannot be reached
ret noret	Consistency of return statements
trd notrd	Possible truncation or loss of data or precision
tru notru	Variable names truncated by the compiler
trx notrx	Hexadecimal floating point constants rounding
uni nouni	Uninitialized variables
use nouse	Unused auto and static variables
vft novft	Generation of virtual function tables

Notes

You can use the **#pragma info** directive to temporarily override the current **-qinfo** compiler option settings specified on the command line, in the configuration file, or by earlier invocations of the **#pragma info** directive.

Example

For example, in the code segments below, the **#pragma info(eff, nouni)** directive preceding MyFunction1 instructs the compiler to generate messages identifying statements or pragmas with no effect, and to suppress messages identifying uninitialized variables. The **#pragma info(restore)** directive preceding MyFunction2 instructs the compiler to restore the message options that were in effect before the **#pragma info(eff, nouni)** directive was invoked.

```
#pragma info(eff, nouni)
int MyFunction1()
{
    .
    .
    .
}

#pragma info(restore)
int MyFunction2()
{
    .
    .
    .
}
```

Related References

“General Purpose Pragmas” on page 297

“info” on page 154

#pragma ishome

C++

Description

The **#pragma ishome** directive informs the compiler that the specified class's home module is the current compilation unit. The home module is where items, such as the virtual function table, are stored. If an item is referenced from outside of the compilation unit, it will not be generated outside its home. The advantage of this is the minimization of code.

Syntax

▶▶ #pragma ishome (—*className*—) ▶▶

where:

className the literal name of the class whose home will be the current compilation unit.

Notes

A warning will be produced if there is a **#pragma ishome** without a matching **#pragma hashome**.

Related References

"General Purpose Pragmas" on page 297

"#pragma hashome" on page 308

#pragma isolated_call

► C ► C++

Description

The `#pragma isolated_call` directive lists a function that does not have or rely on side effects, other than those implied by its parameters.

Syntax

```
► #pragma isolated_call (—function—) ◀
```

where *function* is a primary expression that can be an identifier, operator function, conversion function, or qualified name. An identifier must be of type function or a typedef of function. If the name refers to an overloaded function, all variants of that function are marked as isolated calls.

Notes

The `-qisolated_call` compiler option has the same effect as this pragma.

The pragma informs the compiler that the function listed does not have or rely on side effects, other than those implied by its parameters. Functions are considered to have or rely on side effects if they:

- Access a volatile object
- Modify an external object
- Modify a static object
- Modify a file
- Access a file that is modified by another process or thread
- Allocate a dynamic object, unless it is released before returning
- Release a dynamic object, unless it was allocated during the same invocation
- Change system state, such as rounding mode or exception handling
- Call a function that does any of the above

Essentially, any change in the state of the runtime environment is considered a side effect. Modifying function arguments passed by pointer or by reference is the only side effect that is allowed. Functions with other side effects can give incorrect results when listed in `#pragma isolated_call` directives.

Marking a function as `isolated_call` indicates to the optimizer that external and static variables cannot be changed by the called function and that pessimistic references to storage can be deleted from the calling function where appropriate. Instructions can be reordered with more freedom, resulting in fewer pipeline delays and faster execution in the processor. Multiple calls to the same function with identical parameters can be combined, calls can be deleted if their results are not needed, and the order of calls can be changed.

The function specified is permitted to examine non-volatile external objects and return a result that depends on the non-volatile state of the runtime environment. The function can also modify the storage pointed to by any pointer arguments passed to the function, that is, calls by reference. Do not specify a function that calls itself or relies on local static storage. Listing such functions in the `#pragma isolated_call` directive can give unpredictable results.

The `-qignprag` compiler option causes aliasing pragmas to be ignored. Use the `-qignprag` compiler option to debug applications containing the `#pragma isolated_call` directive.

Example

The following example shows the use of the `#pragma isolated_call` directive. Because the function `this_function` does not have side effects, a call to it will not change the value of the external variable `a`. The compiler can assume that the argument to `other_function` has the value 6 and will not reload the variable from memory.

```
int a;

// Assumed to have no side effects
int this_function(int);

#pragma isolated_call(this_function)
that_function()
{
    a = 6;
    // Call does not change the value of "a"
    this_function(7);

    // Argument "a" has the value 6
    other_function(a);
}
```

Related References

“General Purpose Pragmas” on page 297

“ignprag” on page 153

“isolated_call” on page 170

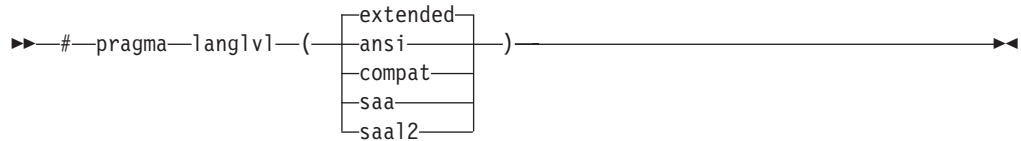
#pragma langlvl

► C ► C++

Description

The *#pragma langlvl* directive selects the C or C++ language level for compilation.

Syntax



where:

- ansi** Defines the predefined macro `__STDC__` and undefines other `langlvl` variables. Allows only language constructs that conform to ANSI/ISO C standards.
- extended** Defines the predefined macro `__EXTENDED__` and undefines other `langlvl` variables. The default language level is extended.
- classic** Defines the predefined macro `__CLASSIC__` and undefines other `langlvl` variables.
- saa** Defines the predefined macro `__SAA__` and undefines other `langlvl` variables. Allows only language constructs that conform to the most recent level of SAA C standards (currently Level 2). These include ANSI C constructs. This language level is valid for C programs only.
- saal2** Defines the predefined macro `__SAAL2__` and undefines other `langlvl` variables. Allows only language constructs that conform to SAA Level 2 C standards. These include ANSI C constructs. This language level is valid for C programs only.

Notes

This pragma can be specified only once in a source file, and it must appear before any statements in a source file.

The compiler uses predefined macros in the header files to make declarations and definitions available that define the specified language level.

This directive can dynamically alter preprocessor behavior. As a result, compiling with the `-E` compiler option may produce results different from those produced when not compiling with the `-E` option.

Related References

“General Purpose Pragas” on page 297

“E” on page 115

“langlvl” on page 175

#pragma leaves

C C++

Description

The **#pragma leaves** directive takes a function name and specifies that the function never returns to the instruction after the call.

Syntax

►► #pragma leaves (*function*) ◄◄

Notes

This pragma tells the compiler that *function* never returns to the caller.

The advantage of the pragma is that it allows the compiler to ignore any code that exists after *function*, in turn, the optimizer can generate more efficient code. This pragma is commonly used for custom error-handling functions, in which programs can be terminated if a certain error is encountered. Some functions which also behave similarly are **exit**, **longjmp**, and **terminate**.

Example

```
#pragma leaves(handle_error_and_quit)
void test_value(int value)
{
    if (value == ERROR_VALUE)
    {
        handle_error_and_quit(value);
        TryAgain(); // optimizer ignores this because
                   // never returns to execute it
    }
}
```

Related References

“General Purpose Pragmas” on page 297

#pragma map

► C ► C++

Description

The **#pragma map** directive tells the compiler that all references to an identifier are to be converted to “*name*”.

Syntax

```
►► #pragma map ( identifier , name )
```

function_signature

where:

<i>identifier</i>	A name of a data object or a nonoverloaded function with external linkage. ► C++ If the identifier is the name of an overloaded function or a member function, there is a risk that the pragma will override the compiler-generated names. This will create problems during linking.
<i>function_signature</i>	A name of a function or operator with internal linkage. The name can be qualified.
<i>name</i>	The external name that is to be bound to the given object, function, or operator. ► C++ Specify the mangled name if linking into a C++ name (a name that will have C++ linkage signature, which is the default signature in C++). See Example 3, in the Examples section below.

Notes

You should not use **#pragma map** to map the following:

- C++ Member functions
- Overloaded functions
- Objects generated from templates
- Functions with built in linkage

The directive can appear anywhere in the program. The identifiers appearing in the directive, including any type names used in the prototype argument list, are resolved as though the directive had appeared at file scope, independent of its actual point of occurrence.

Examples

Example 1 ► C

```
int funcname1()
{
    return 1;
}

#pragma map(func , "funcname1") // maps ::func to funcname1

int main()
{
    return func();           // no function prototype needed in C
}
```

Example 2 C++

```
extern "C" int funcname1()
{
    return 0;
}

extern "C" int func(); //function prototypes needed in C++

#pragma map(func , "funcname1") // maps ::func to funcname1

int main()
{
    return func();
}
```

Example 3 C++

```
#pragma map(foo, "bar__Fv")

int foo(); //function prototypes needed in C++

int main()
{
    return foo();
}

int bar() {return 7;}
```

Note: You can avoid using the mangled name **bar_FV** by declaring **bar** as having C linkage. See Example 4, below.

Example 4 C++

```
#pragma map(foo, "bar")

int foo(); //function prototypes needed in C++

int main()
{
    return foo();
}

extern "C" int bar() {return 7;}
```

Related References

“General Purpose Pragmas” on page 297

#pragma mc_func

► C ► C++

Description

The `#pragma mc_func` directive allows you to specify machine instructions for a particular function.

Syntax

►► #pragma mc_func *function* { *literal* } ►►

where:

function

- Is a previously undeclared function with no parameters and a return type of `int`. This will declare the function.

literal

- Is a previously declared function with a return type of `int`.
- a string that contains zero or more hexadecimal digits. The number of digits must be even.

Related References

“General Purpose Pragmas” on page 297

“#pragma reg_killed_by” on page 338

#pragma namemangling

C++

Description

The **#pragma namemangling** directive sets the name mangling scheme and the maximum length of external symbol names generated from C++ source code.

Syntax

```
▶▶ #pragma namemangling ( [ansi|compat] [ , -num_chars ] ) ▶▶
```

where:

- ansi The name mangling scheme fully supports the various language features of Standard C++, including function template overloading. If you specify **ansi** but do not specify a size with *num_chars*, the default maximum is 64000 characters.
- v5 The name mangling scheme is compatible with VisualAge C++ version 4.0. If you specify **v5** but do not specify a size with *num_chars*, the default maximum is 64000 characters.
- v4 The name mangling scheme is compatible with VisualAge C++ version 4.0. If you specify **v4** but do not specify a size with *num_chars*, the default maximum is 64000 characters.
- v3 The name mangling scheme is compatible with versions of VisualAge C++ earlier than version v4.0. If you specify **v3** but do not specify a size with *num_chars*, the default maximum is 255 characters. Use this scheme for compatibility with link modules created with versions of VisualAge C++ released prior to version 4.0, or with link modules that were created with the **#pragma namemangling** or **-qnamemangling=compat** compiler options specified.
- compat Same as the **v3** suboption, described above.

Related References

“General Purpose Pragmas” on page 297

“namemangling” on page 214

“#pragma nameManglingRule” on page 323

#pragma nameManglingRule

C++

Description

The **#pragma nameManglingRule** directive instructs the compiler whether or not to mangle function names according to their function parameter types.

Syntax

```
▶▶ #pragma nameManglingRule (fnparmtype, 

|     |
|-----|
| on  |
| off |
| pop |

)
```

where:

- on Function arguments are mangled according to function parameter types. For example, cv qualifiers in function arguments are not mangled..
- off Name mangling is compatible with VisualAge C++ version 5.0, and cv qualifiers in function arguments are mangled.
- pop Discards the current **#pragma nameManglingRule** setting, and replaces it with the previous **#pragma nameManglingRule** setting from the stack. If no previous settings remain on the stack, the default **#pragma nameManglingRule** setting is used.

Defaults

The default is **#pragma nameManglingRule(fnparmtype, on)** when the **-qnamemangling=ansi** or **#pragma namemangling(ansi)** compiler options are in effect.

For all other settings of **-qnamemangling** or **#pragma namemangling**, the default is **#pragma nameManglingRule(fnparmtype, off)**.

Notes

This directive provides name mangling scheme compatibility between the current level of the VisualAge C++ compiler and previous versions.

#pragma nameManglingRule is allowed in global, class, and function scopes. Different pragma settings can be specified in front of function declarations and definitions. If **#pragma nameManglingRule** settings in subsequent declarations and definitions conflict, the compiler ignores those settings and issues a warning message.

A given **#pragma nameManglingRule** setting remains in effect until overridden by another **#pragma nameManglingRule** setting.

Each new **#pragma nameManglingRule** setting is pushed onto a stack, over top of the previously-specified setting. The setting currently in effect can be removed from the top of the stack with the **pop** suboption. It is replaced by the previous setting stored in the stack, if any remain. If no settings remain, the default **#pragma nameManglingRule** setting is used.

Related References

- “General Purpose Pragmas” on page 297
- “namemangling” on page 214
- “#pragma namemangling” on page 322

#pragma object_model

➤ C++

Description

The **#pragma object_model** directive specifies the object model to use for the structures, unions, and classes that follow it.

Syntax

```
➤ #pragma object_model ( [ compat  
                        | ibm  
                        | pop ] ) ➤
```

where choices for object model are:

- compat Uses the x1C object model compatible with previous versions of the compiler.
- ibm Uses the new object model.
- pop Reverts to the object model setting previously in effect. If no previous object model setting exists, sets the object model to the default setting..

Notes

This pragma affects the structures, unions, and classes that follow it, until another **#pragma objmodel** statement is reached.

When this pragma is used, the current object model setting is placed on a stack. Subsequent use of this pragma places the newest setting on top of the stack. Specifying **#pragma objmodel(pop)** removes the current object model setting from the stack, and sets the object model to the next setting in the stack.

Related Concepts

“Object Models” on page 4

Related References

“General Purpose Pragmas” on page 297

“objmodel” on page 220

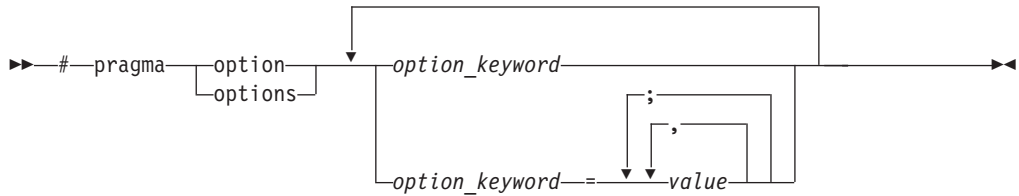
#pragma options

C C++

Description

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

Syntax



Notes

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

To specify more than one compiler option with the **#pragma options** directive,, separate the options using a blank space. For example:

```
#pragma options langlvl=ansi halt=s spill=1024 source
```

Most **#pragma options** directives must come before any statements in your source program; only comments, blank lines, and other **#pragma** specifications can precede them. For example, the first few lines of your program can be a comment followed by the **#pragma options** directive:

```

/* The following is an example of a #pragma options directive: */
#pragma options langlvl=ansi halt=s spill=1024 source
/* The rest of the source follows ... */
  
```

Options specified before any code in your source program apply to your entire program source code. You can use other **#pragma** directives throughout your program to turn an option on for a selected block of source code. For example, you can request that parts of your source code be included in your compiler listing:

```

#pragma options source
/* Source code between the source and nosource #pragma
   options is included in the compiler listing */
#pragma options nosource
  
```

The settings in the table below are valid *options* for **#pragma options**. For more information, refer to the pages for the equivalent compiler option.

Language Application	Valid settings for #pragma options option_keyword	Compiler option equivalent	Description
C C++	align=option	-qalign	Specifies what aggregate alignment rules the compiler uses for file compilation.

Language Application		Valid settings for #pragma options <i>option_keyword</i>	Compiler option equivalent	Description
> C	> C++	[no]ansialias	-qansialias	Specifies whether type-based aliasing is to be used during optimization.
> C	> C++	arch= <i>option</i>	-qarch	Specifies the architecture on which the executable program will be run.
> C	> C++	assert= <i>option</i>	-qassert	Requests the compiler to apply aliasing assertions to your compilation unit.
> C	> C++	[no]attr attr=full	-qattr	Produces an attribute listing containing all names.
> C	> C++	chars= <i>option</i>	-qchars See also #pragma chars	Instructs the compiler to treat all variables of type char as either signed or unsigned.
> C	> C++	[no]check	-qcheck	Generates code which performs certain types of run-time checking.
> C	> C++	[no]compact	-qcompact	When used with optimization, reduces code size where possible, at the expense of execution speed.
> C	> C++	[no]dbcs	-qmbcs, dbcs	String literals and comments can contain DBCS characters.
> C		[no]dbxextra	-qdbxextra	Generates symbol table information for unreferenced variables.
> C	> C++	[no]digraph	-qdigraph	Allows special digraph and keyword operators.
> C	> C++	[no]dollar	-qdollar	Allows the \$ symbol to be used in the names of identifiers.
> C	> C++	enum= <i>option</i>	-qenum See also #pragma enum	Specifies the amount of storage occupied by the enumerations.
> C	> C++	[no]extchk	-qextchk	Performs external name type-checking and function call checking.

Language Application		Valid settings for #pragma options <i>option_keyword</i>	Compiler option equivalent	Description
> C	> C++	flag= <i>option</i>	-qflag	Specifies the minimum severity level of diagnostic messages to be reported. Severity levels can also be specified with: #pragma options flag=i => #pragma report (level,I) #pragma options flag=w => #pragma report (level,W) #pragma options flag=e,s,u => #pragma report (level,E)
> C	> C++	float=[no] <i>option</i>	-qfloat	Specifies various floating point options to speed up or improve the accuracy of floating point operations.
> C	> C++	[no]flttrap= <i>option</i>	-qflttrap	Generates extra instructions to detect and trap floating point exceptions.
> C	> C++	[no]fold	-qfold	Specifies that constant floating point expressions are to be evaluated at compile time.
> C	> C++	[no]fullpath	-qfullpath	Specifies the path information stored for files for dbx stabstrings.
> C	> C++	[no]funcsect	-qfuncsect	Places instructions for each function in a separate cset.
> C	> C++	halt	-qhalt	Stops compiler when errors of the specified severity detected.
> C	> C++	[no]idirfirst	-qidirfirst	Specifies search order for user include files.
> C	> C++	[no]ignerrno	-qignerrno	Allows the compiler to perform optimizations that assume errno is not modified by system calls.
> C	> C++	[no]ignprag	-qignprag	Instructs the compiler to ignore certain pragma statements.
> C	> C++	[no]info= <i>option</i>	-qinfo See also #pragma info	Produces informational messages.
> C	> C++	initauto= <i>value</i>	-qinitauto	Initializes automatic storage to a specified hexadecimal byte value.

Language Application		Valid settings for #pragma options <i>option_keyword</i>	Compiler option equivalent	Description
> C	> C++	[no]inlglue	-qinlglue	Generates fast external linkage by inlining the pointer glue code necessary to make a call to an external function or a call through a function pointer.
> C	> C++	isolated_call= <i>names</i>	-qisolated_call See also #pragma isolated_call	Specifies functions in the source file that have no side effects.
> C	> C++	langlvl	-qlanglvl	Specifies different language levels. This directive can dynamically alter preprocessor behavior. As a result, compiling with the -E compiler option may produce results different from those produced when not compiling with the -E option.
> C	> C++	[no]dbl128	-qdbl128, longdouble	Increases the size of long double type from 64 bits to 128 bits.
> C	> C++	[no]libansi	-qlibansi	Assumes that all functions with the name of an ANSI C library function are in fact the system functions.
> C	> C++	[no]list	-qlist	Produces a compiler listing that includes an object listing.
> C	> C++	[no]longlong	-qlonglong	Allows long long types in your program.
> C		[no]macpstr	-qmacpstr	Converts Pascal string literals into null-terminated strings where the first byte contains the length of the string.
> C	> C++	[no]maf	-qmaf	Specifies whether floating-point multiply-add instructions are to be generated.
> C	> C++	[no]maxmem= <i>number</i>	-qmaxmem	Instructs the compiler to halt compilation when a specified number of errors of specified or greater severity is reached.
> C	> C++	[no]mbcs	-qmbcs, dbcs	String literals and comments can contain DBCS characters.

Language Application	Valid settings for #pragma options <i>option_keyword</i>	Compiler option equivalent	Description
> C++	priority= <i>number</i>	-qpriority See also “#pragma priority” on page 336	Specifies the priority level for the initialization of static constructors
> C > C++	[no]proclocal, [no]procimported, [no]procunknown	-qproclocal, procimported, procunknown	Marks functions as local, imported, or unknown.
> C	[no]proto	-qproto	If this option is set, the compiler assumes that all functions are prototyped.
> C > C++	[no]rndsngl	-qrndsngl	Specifies that the results of each single-precision float operation is to be rounded to single precision.
> C > C++	[no]ro	-qro	Specifies the storage type for string literals.
> C > C++	[no]roconst	-qroconst	Specifies the storage location for constant values.
> C > C++	[no]rrm	-qrrm	Prevents floating-point optimizations that are incompatible with run-time rounding to plus and minus infinity modes.
> C > C++	[no]showinc	-qshowinc	If used with -qsource , all include files are included in the source listing.
> C > C++	[no]source	-qsource	Produces a source listing.
> C > C++	spill= <i>number</i>	-qspill	Specifies the size of the register allocation spill area.
> C	[no]srcmsg	-qsrcmsg	Adds the corresponding source code lines to the diagnostic messages in the stderr file.
> C > C++	[no]stdinc	-qstdinc	Specifies which files are included with #include <file_name> and #include “file_name” directives.
> C > C++	[no]strict	-qstrict	Turns off aggressive optimizations of the -O3 compiler option that have the potential to alter the semantics of your program.
> C > C++	tftable= <i>option</i>	-qtftable	Changes the length of tabs as perceived by the compiler.

Language Application		Valid settings for #pragma options <i>option_keyword</i>	Compiler option equivalent	Description
> C	> C++	tune= <i>option</i>	-qtune	Specifies the architecture for which the executable program is optimized.
> C	> C++	[no]unroll unroll= <i>number</i>	-qunroll	Unrolls inner loops in the program by a specified factor.
> C		[no]upconv	-qupconv	Preserves the unsigned specification when performing integral promotions.
	> C++	[no]vftable	-qvftable	Controls the generation of virtual function tables.
> C	> C++	[no]xref	-qxref	Produces a compiler listing that includes a cross-reference listing of all identifiers.

Related References

“General Purpose Pragmas” on page 297

“E” on page 115

#pragma option_override

► C ► C++

Description

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

Syntax

```
► #pragma option_override ( func_name [, "option"] ) ◀
```

Notes

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

Per-function optimizations have effect only if optimization is already enabled by compilation option. You can request per-function optimizations at a level less than or greater than that applied to the rest of the program being compiled. Selecting options through this pragma affects only the specific optimization option selected, and does not affect the implied settings of related options.

Options are specified in double quotes, so they are not subject to macro expansion. The option specified within quotes must comply with the syntax of the build option.

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

This pragma affects only functions defined in your compilation unit and can appear anywhere in the compilation unit, for example:

- before or after a compilation unit
- before or after the function definition
- before or after the function declaration
- before or after a function has been referenced
- inside or outside a function definition.

Related References

“General Purpose Pragmas” on page 297

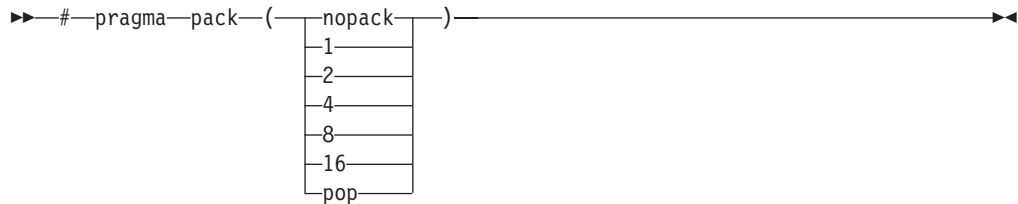
#pragma pack

C C++

Description

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

Syntax



where:

- | | |
|-----------------------|--------------------------------------------------------------------|
| 1 2 4 8
 16 | Members of structures are aligned on the specified byte-alignment. |
| nopack | No packing is applied, and "nopack" is pushed onto the pack stack |
| pop | The top element on the pragma pack stack is popped. |

Notes

The **#pragma pack** directive modifies the current alignment rule for only the members of structures whose declarations follow the directive. It does not affect the alignment of the structure directly, but by affecting the alignment of the members of the structure, it may affect the alignment of the overall structure according to the alignment rule.

The **#pragma pack** directive cannot increase the alignment of a member, but rather can decrease the alignment. For example, for a member with data type of integer (int), a **#pragma pack(2)** directive would cause that member to be packed in the structure on a 2-byte boundary, while a **#pragma pack(4)** directive would have no effect.

The **#pragma pack** directive is stack based. All pack values are pushed onto a stack as the source code is parsed. The value at the top of the current pragma pack stack is the value used to pack members of all subsequent structures within the scope of the current alignment rule.

A **#pragma pack** stack is associated with the current element in the alignment rule stack. Alignment rules are specified with the **-qalign** compiler option or with the **#pragma options align** directive. If a new alignment rule is created, a new **#pragma pack** stack is created. If the current alignment rule is popped off the alignment rule stack, the current **#pragma pack** stack is emptied and the previous **#pragma pack** stack is restored. Stack operations (pushing and popping pack settings) affect only the current **#pragma pack** stack.

The pragma pack directive does not affect the alignment of the bits in a bitfield.

Examples

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

```
struct s_t1 {
    char a;
    int b;
    #pragma pack(1)
    struct s_t2 {
        char x;
        int y;
    } S2;
    char c;
    int b;
} S1;
```

2. In the code segment below:
 - a. The members of S1 would be aligned with the twobyte alignment rule, S2 members would be packed on 1-byte, S3 members packed on 2-bytes, and S4 packed on 4-bytes.
 - b. The **#pragma options align=reset** directive pops the current alignment rule (which in the above case was the twobyte alignment rule). All **#pragma pack** directives issued while the **#pragma options align=twobyte** directive was in effect are also popped.
 - c. The **#pragma pack(4)** directive encountered is restored, as well as the alignment rule that was in effect before the **#pragma options align=twobyte** directive was popped.

```
#pragma pack(4)
#pragma options align=twobyte
struct s_t1 {
    char a;
    int b;
} S1;

#pragma pack(1)
struct s_t2 {
    char a;
    short b;
} S2;

#pragma pack(2)
struct s_t3 {
    char a;
    double b;
} S3;

#pragma options align=reset
struct s_t4 {
    char a;
    int b;
} S4;
```

3. This example shows how a **#pragma pack** directive can affect the size and mapping of a structure:

```
struct s_t {
    char a;
    int b;
    short c;
    int d;
} S;
```

Default mapping:

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

With #pragma pack(1):

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

Related References

“General Purpose Pragmas” on page 297

“align” on page 77

“#pragma options” on page 325

#pragma pass_by_value

C++

Description

The **#pragma pass_by_value** directive specifies how classes containing const or reference members are passed in function arguments. All classes in the compilation unit are affected by this option.

Syntax

```
▶▶ #pragma pass_by_value ( compat ) ▶▶
```

ansi
default
source
pop
reset

where:

compat	Pushes the equivalent of the -qoldpassbyvalue on to the stack.
ansi	Pushes the equivalent of the -qnooldpassbyvalue option on to the stack.
default	Pushes the compiler default setting for the option -qnooldpassbyvalue on to the stack.
source	Pushes the value of the original command line option (-qoldpassbyvalue or -qnooldpassbyvalue) on to the stack.
pop	Pops the stack, discarding the current #pragma pass_by_value setting, and restoring the previous #pragma pass_by_value setting on the stack as the option now in effect.
reset	Same as pop .

Notes

The current setting of **#pragma pass_by_value** specifies how classes are passed as function arguments in the program source code following the pragma. The setting remains in effect until a subsequent use of the pragma invokes a new setting.

The setting of **#pragma pass_by_value** overrides the **-qoldpassbyvalue** compiler option.

If **pop** or **reset** is called on an empty stack, the compiler issues a warning message and assumes the **-qoldpassbyvalue** setting originally set on the command line. If **-qoldpassbyvalue** was not set on the command line, the compiler will assume the default setting set in the compiler default configuration file.

Use **#pragma pass_by_value(compat)** to instruct the compiler to mimic the behavior of earlier VisualAge C++ compilers (v3.6 or earlier) when passing classes as function arguments. Classes containing a const or reference member are not passed by value.

Related References

“General Purpose Pragmas” on page 297

“oldpassbyvalue” on page 221

#pragma priority

C++

Description

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

Syntax

▶▶ #pragma priority(*n*) ▶▶

Notes

Where *n* is an integer literal in the range of INT_MIN to INT_MAX. The default value is 0. A negative value indicates a higher priority; a positive value indicates a lower priority. The first 1024 priorities (INT_MIN to INT_MIN + 1023) are reserved for use by the compiler and its libraries. The priority value specified applies to all runtime static initialization in the current compilation unit.

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

For example, if the constructor to object B is passed object A as a parameter, then VisualAge C++ will arrange for A to be constructed first, even if this violates the top-to-bottom or #pragma priority ordering. This is essential for orderless programming, which VisualAge C++ permits. If the target is an .obj/.lib, this processing is not done, because there may not be enough information to detect the dependencies.

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

Example

```
#pragma priority(1)
```

Related References

“General Purpose Pragmas” on page 297

#pragma reachable

► C ► C++

Description

The **#pragma reachable** directive declares that the point after the call to a routine, *function*, can be the target of a branch from some unknown location. This pragma should be used in conjunction with `setjmp`.

Syntax

► #pragma reachable (*function*) ◀

Related References

“General Purpose Pragmas” on page 297

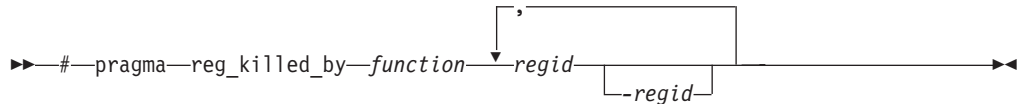
#pragma reg_killed_by

C C++

Description

The **#pragma reg_killed_by** directive specifies those registers whose value will be corrupted by the specified function. The list of registers that follow the function name will become the list of registers killed by the function. This #pragma can only be used on functions that are defined using **#pragma mc_func**.

Syntax



where:

function The function previously defined using the **#pragma mc_func**.
regid Either a single register or the beginning and ending registers in a range.

Notes

A single register is volatile according to the register conventions. *regid* is subject to the following restrictions:

- the class name part of the register name must be valid
- the register number is either required or prohibited
- when the register number is required it must be in the valid range

If any of these restrictions are not met, an error is issued and the register is ignored.

Example

```
#pragma reg_killed_by function_a fp0-fp31
```

Related References

“General Purpose Pragmas” on page 297
“#pragma mc_func” on page 321

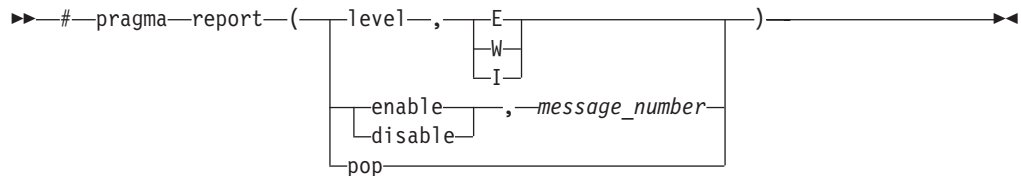
#pragma report

C++

Description

The **#pragma report** directive controls the generation of specific messages. The pragma will take precedence over **#pragma info**. Specifying **#pragma report(pop)** will revert the report level to the previous level. If no previous report level was specified, then a warning will be issued and the report level will remain unchanged.

Syntax



where:

level	Indicates the minimum severity level of diagnostic messages to display.
E W I	Used in conjunction with level to determine the type of diagnostic messages to display.
E	Signifies a minimum message severity of 'error'. This is considered as the most severe type of diagnostic message. A report level of 'E' will display only 'error' messages. An alternative way of setting the report level to 'E' is by specifying the -qflag(E) compiler option.
W	Signifies a minimum message severity of 'warning'. A report level of 'W' will filter out all informational messages, and display only those messages classified as warning or error messages. An alternative way of setting the report level to 'W' is by specifying the -qflag(E) compiler option.
I	Signifies a minimum message severity of 'information'. Information messages are considered as the least severe type of diagnostic message. A level of 'I' would display messages of all types. The VisualAge C++ development environment sets this as the default option. An alternative way of setting the report level to 'I' is by specifying the -qflag(E) compiler option.
enable disable	Enables or disables the specified message number.
message_number	Is an identifier containing the message number prefix, followed by the message number. An example of a message number is: CPPC1004
pop	resets the report level back to the previous report level. If a pop operation is performed on an empty stack, the report level will remain unchanged and no message will be generated.

Examples

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

```
1 #pragma info(all)
2 #pragma report(level, W)
```

2. If CPPC1000 was an error message, it would be displayed. If it was any other type of diagnostic message, it would not be displayed.

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

Changing the order of the code like so:

```
1 #pragma report(level, E)
2 #pragma report(enable, CPPC1000)
```

would yield the same result. The order in which the two lines of code appear in, does not affect the outcome. However, if the message was 'disabled', then regardless of what report level is set and order the lines of code appear in, the diagnostic message will not be displayed.

3. In line 1 of the example below, the initial report level is set to 'I', causing message CPPC1000 to display regardless of the type of diagnostic message it classified as. In line 3, a new report level of 'E' is set, indicating that only messages with a severity level of 'E' will be displayed. Immediately following line 3, the current level 'E' is 'popped' and reset back to 'I'.

```
1 #pragma report(level, I)
2 #pragma report(enable, CPPC1000)
3 #pragma report(level, E)
4 #pragma report(pop)
```

Related References

"General Purpose Pragmas" on page 297

"flag" on page 130

#pragma strings

► C ► C++

Description

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

Syntax

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

Notes

Strings are read-only by default.

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

Example

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

Related References

“General Purpose Pragmas” on page 297

#pragma unroll

C C++

Description

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

Syntax

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

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

Notes

The **#pragma unroll** and **#pragma nounroll** directives must appear immediately before the loop to be affected. Only one of these directives can be specified for a given loop.

Specifying **#pragma nounroll** for a loop instructs the compiler to not unroll that loop. Specifying **#pragma unroll(1)** has the same effect.

To see if the **unroll** option improves performance of a particular application, you should first compile the program with usual options, then run it with a representative workload. You should then recompile with command line **-qunroll** option and/or the **unroll** pragmas enabled, then rerun the program under the same conditions to see if performance improves.

Examples

1. In the following example, loop control is not modified:

```
#pragma unroll(2)
while (*s != 0)
{
    *p++ = *s++;
}
```

Unrolling this by a factor of 2 gives:

```
while (*s)
{
    *p++ = *s++;
    if (*s == 0) break;
    *p++ = *s++;
}
```

2. In this example, loop control *is* modified:

```
#pragma unroll(3)
for (i=0; i<n; i++) {
    a[i]=b[i] * c[i];
}
```

Unrolling by 3 gives:

```
i=0;
if (i>n-2) goto remainder;
for (; i<n-2; i+=3) {
    a[i]=b[i] * c[i];
```

```
    a[i+1]=b[i+1] * c[i+1];
    a[i+2]=b[i+2] * c[i+2];
}
if (i<n) {
    remainder:
    for (; i<n; i++) {
        a[i]=b[i] * c[i];
    }
}
```

Related References

“General Purpose Pragmas” on page 297

“unroll” on page 283

Pragmas to Control Parallel Processing

The `#pragma` directives on this page give you control over how the compiler handles parallel processing in your program. These pragmas fall into two groups; IBM-specific directives, and directives conforming to the OpenMP Application Program Interface specification.

Use the `-qsmp` compiler option to specify how you want parallel processing handled in your program. You can also instruct the compiler to ignore all parallel processing-related `#pragma` directives by specifying the `-qignprag=ibm:omp` compiler option.

Directives apply only to the statement or statement block immediately following the directive.

IBM Pragma Directives ▶ C	Description
<code>#pragma ibm critical</code>	Instructs the compiler that the statement or statement block immediately following this pragma is a critical section.
<code>#pragma ibm independent_calls</code>	Asserts that specified function calls within the chosen loop have no loop-carried dependencies.
<code>#pragma ibm independent_loop</code>	Asserts that iterations of the chosen loop are independent, and that the loop can therefore be parallelized.
<code>#pragma ibm iterations</code>	Specifies the approximate number of loop iterations for the chosen loop.
<code>#pragma ibm parallel_loop</code>	Explicitly instructs the compiler to parallelize the chosen loop.
<code>#pragma ibm permutation</code>	Asserts that specified arrays in the chosen loop contain no repeated values.
<code>#pragma ibm schedule</code>	Specifies scheduling algorithms for parallel loop execution.
<code>#pragma ibm sequential_loop</code>	Explicitly instructs the compiler to execute the chosen loop sequentially.

OpenMP Pragma Directives ▶ C ▶ C++	Description
<code>#pragma omp atomic</code>	Identifies a specific memory location that must be updated atomically and not be exposed to multiple, simultaneous writing threads.
<code>#pragma omp parallel</code>	Defines a parallel region to be run by multiple threads in parallel. With specific exceptions, all other OpenMP directives work within parallelized regions defined by this directive.
<code>#pragma omp for</code>	Work-sharing construct identifying an iterative for-loop whose iterations should be run in parallel.
<code>#pragma omp parallel for</code>	Shortcut combination of omp parallel and omp for pragma directives, used to define a parallel region containing a single for directive.
<code>#pragma omp ordered</code>	Work-sharing construct identifying a structured block of code that must be executed in sequential order.

OpenMP Pragma Directives C C++	Description
#pragma omp section, #pragma omp sections	Work-sharing construct identifying a non-iterative section of code containing one or more subsections of code that should be run in parallel.
#pragma omp parallel sections	Shortcut combination of omp parallel and omp sections pragma directives, used to define a parallel region containing a single sections directive.
#pragma omp single	Work-sharing construct identifying a section of code that must be run by a single available thread.
#pragma omp master	Synchronization construct identifying a section of code that must be run only by the master thread.
#pragma omp critical	Synchronization construct identifying a statement block that must be executed by a single thread at a time.
#pragma omp barrier	Synchronizes all the threads in a parallel region.
#pragma omp flush	Synchronization construct identifying a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory.
#pragma omp threadprivate	Defines the scope of selected file-scope data variables as being private to a thread, but file-scope visible within that thread.

Related Concepts

“Program Parallelization” on page 9

Related Tasks

“Set Parallel Processing Run-time Options” on page 20

“Control Parallel Processing with Pragmas” on page 45

Related References

“smp” on page 252

“IBM SMP Run-time Options for Parallel Processing” on page 383

“OpenMP Run-time Options for Parallel Processing” on page 386

“Built-in Functions Used for Parallel Processing” on page 388

For complete information about the OpenMP Specification, see:

OpenMP Web site

OpenMP Specification.

#pragma ibm critical



Description

The **critical** pragma identifies a critical section of program code that must only be run by one process at a time.

Syntax

```
#pragma ibm critical [(name)]  
<statement>
```

where *name* can be used to optionally identify the critical region. Identifiers naming a critical region have external linkage.

Notes

The compiler reports an error if you try to branch into or out of a critical section. Some situations that will cause an error are:

- A critical section that contains the **return** statement.
- A critical section that contains **goto**, **continue**, or **break** statements that transfer program flow outside of the critical section.
- A **goto** statement outside a critical section that transfers program flow to a label defined within a critical section.

Related References

“Pragmas to Control Parallel Processing” on page 344

#pragma ibm independent_calls

► C

Description

The `independent_calls` pragma asserts that specified function calls within the chosen loop have no loop-carried dependencies. This information helps the compiler perform dependency analysis.

Syntax

```
#pragma ibm independent_calls [(identifier [,identifier] ... )]  
<countable for/while/do loop>
```

where *identifier* represents the name of a function.

Notes

identifier cannot be the name of a pointer to a function.

If no function identifiers are specified, the compiler assumes that all functions inside the loop are free of carried dependencies.

Related References

“Pragmas to Control Parallel Processing” on page 344

#pragma ibm independent_loop



Description

The **independent_loop** pragma asserts that iterations of the chosen loop are independent, and that the loop can be parallelized.

Syntax

```
#pragma ibm independent_loop [if (exp)]  
<countable for/while/do loop>
```

where *exp* represents a scalar expression.

Notes

When the **if** argument is specified, loop iterations are considered independent only as long as *exp* evaluates to TRUE at run-time.

This pragma can be combined with the **schedule** pragma to select a specific parallel process scheduling algorithm. For more information, see the description for the **schedule** pragma.

Related References

“Pragmas to Control Parallel Processing” on page 344

“#pragma ibm schedule” on page 352

#pragma ibm iterations



Description

The **iterations** pragma specifies the approximate number of loop iterations for the chosen loop.

Syntax

```
#pragma ibm iterations (iteration-count)  
<countable for/while/do loop>
```

where *iteration-count* represents a positive integral constant expression.

Notes

The compiler uses the information in the *iteration-count* variable to determine if it is efficient to parallelize the loop.

Related References

“Pragmas to Control Parallel Processing” on page 344

#pragma ibm parallel_loop



Description

The **parallel_loop** pragma explicitly instructs the compiler to parallelize the chosen loop.

Syntax

```
#pragma ibm parallel_loop [if (exp)] [schedule (sched-type)]  
<countable for/while/do loop>
```

where *exp* represents a scalar expression, and *sched-type* represents any scheduling algorithm as valid for the *schedule* directive.

Notes

When the *if* argument is specified, the loop executes in parallel only if *exp* evaluates to TRUE at run-time. Otherwise the loop executes sequentially. The loop will also run sequentially if it is in a critical section.

This pragma can be applied to a wide variety of C loops, and the compiler will try to determine if a loop is countable or not.

Program sections using the **parallel_loop** pragma must be able to produce a correct result in both sequential and parallel mode. For example, loop iterations must be independent before the loop can be parallelized. Explicit parallel programming techniques involving condition synchronization are not permitted.

This pragma can be combined with the **schedule** pragma to select a specific parallel process scheduling algorithm. For more information, see the description for the **schedule** pragma.

A warning is generated if this pragma is not followed by a countable loop.

Related References

“Pragmas to Control Parallel Processing” on page 344

“#pragma ibm schedule” on page 352

#pragma ibm permutation

► C

Description

The **permutation** pragma asserts that specified arrays in the chosen loop contain no repeated values.

Syntax

```
#pragma ibm permutation (identifier [,identifier] ... )  
<countable for/while/do loop>
```

where *identifier* represents the name of an array.

Notes

identifier cannot be the name of a pointer or a variable modified type.

An array specified by this pragma cannot be a function parameter.

Related References

“Pragmas to Control Parallel Processing” on page 344

#pragma ibm schedule



Description

The `schedule` pragma specifies the scheduling algorithms used for parallel processing.

Syntax

```
#pragma ibm schedule (sched-type)  
<countable for/while/do loop>
```

where *sched-type* represents one of the following options:

affinity	Iterations of a loop are initially divided into local partitions of size ceiling (<i>number_of_iterations/number_of_threads</i>). Each local partition then further subdivided into chunks of size ceiling (<i>number_of_iterations_remaining_in_partition/2</i>). When a thread becomes available, it takes the next chunk from its local partition. If there are no more chunks in the local partition, the thread takes an available chunk from the partition of another thread.
affinity, <i>n</i>	As above, except that each local partition is subdivided into chunks of size <i>n</i> . <i>n</i> must be an integral assignment expression of value 1 or greater.
dynamic	Iterations of a loop are divided into chunks of size 1. Chunks are assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.
dynamic, <i>n</i>	As above, except that all chunks are set to size <i>n</i> . <i>n</i> must be an integral assignment expression of value 1 or greater.
guided	Chunks are made progressively smaller until a chunk size of one is reached. The first chunk is of size ceiling (<i>number_of_iterations/number_of_threads</i>). Remaining chunks are of size ceiling (<i>number_of_iterations_remaining/number_of_threads</i>). Chunks are assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.
guided, <i>n</i>	As above, except the minimum chunk size is set to <i>n</i> . <i>n</i> must be an integral assignment expression of value 1 or greater.
runtime	Scheduling policy is determined at run-time.
static	Iterations of a loop are divided into chunks of size ceiling (<i>number_of_iterations/number_of_threads</i>). Each thread is assigned a separate chunk. This scheduling policy is also known as <i>block scheduling</i> .
static, <i>n</i>	Iterations of a loop are divided into chunks of size <i>n</i> . Each chunk is assigned to a thread in <i>round-robin</i> fashion. <i>n</i> must be an integral assignment expression of value 1 or greater. This scheduling policy is also known as <i>block cyclic scheduling</i> .
static,1	Iterations of a loop are divided into chunks of size 1. Each chunk is assigned to a thread in <i>round-robin</i> fashion. This scheduling policy is also known as <i>cyclic scheduling</i> .

Notes

Scheduling algorithms for parallel processing can be specified using any of the methods shown below. If used, methods higher in the list override entries lower in the list.

- pragma statements
- compiler command line options
- run-time command line options
- run-time default options

Scheduling algorithms can also be specified using the **schedule** argument of the **parallel_loop** and **independent_loop** pragma statements. For example, the following sets of statements are equivalent:

```
#pragma ibm parallel_loop  
#pragma ibm schedule (sched_type)  
<countable for|while|do loop>  
and  
#pragma ibm parallel_loop (sched_type)  
<countable for|while|do loop>
```

If different scheduling types are specified for a given loop, the last one specified is applied.

Related References

“Pragmas to Control Parallel Processing” on page 344

“#pragma ibm independent_loop” on page 348

“#pragma ibm parallel_loop” on page 350

#pragma ibm sequential_loop



Description

The **sequential_loop** pragma explicitly instructs the compiler to execute the chosen loop sequentially.

Syntax

```
#pragma ibm sequential_loop  
<countable for/while/do loop>
```

Notes

This pragma disables automatic parallelization of the chosen loop, and is always respected by the compiler.

Related References

“Pragmas to Control Parallel Processing” on page 344

#pragma omp atomic

C C++

Description

The **omp atomic** directive identifies a specific memory location that must be updated atomically and not be exposed to multiple, simultaneous writing threads.

Syntax

```
#pragma omp atomic
<statement_block>
```

where *statement* is an expression statement of scalar type that takes one of the forms that follow:

<i>statement</i>	Conditions
$x \text{ bin_op} = \text{expr}$	where: <i>bin_op</i> is one of: + * - / & ^ << >> <i>expr</i> is an expression of scalar type that does not reference <i>x</i> .
$x++$	
$++x$	
$x--$	
$--x$	

Notes

Load and store operations are atomic only for object *x*. Evaluation of *expr* is not atomic.

All atomic references to a given object in your program must have a compatible type.

Objects that can be updated in parallel and may be subject to race conditions should be protected with the **omp atomic** directive.

Examples

```
extern float x[], *p = x, y;
/* Protect against race conditions among multiple updates. */
#pragma omp atomic
x[index[i]] += y;
/* Protect against races with updates through x. */
#pragma omp atomic
p[i] -= 1.0f;
```

Related References

“Pragmas to Control Parallel Processing” on page 344

#pragma omp parallel

Description

The **omp parallel** directive explicitly instructs the compiler to parallelize the chosen segment of code.

Syntax

```
#pragma omp parallel [clause[ clause] ...]  
<statement_block>
```

where *clause* is any of the following:

<i>if</i> (<i>exp</i>)	When the <i>if</i> argument is specified, the program code executes in parallel only if the scalar expression represented by <i>exp</i> evaluates to a non-zero value at run-time. Only one if clause can be specified.
<i>private</i> (<i>list</i>)	Declares the scope of the data variables in <i>list</i> to be private to each thread. Data variables in <i>list</i> are separated by commas.
<i>firstprivate</i> (<i>list</i>)	Declares the scope of the data variables in <i>list</i> to be private to each thread. Each new private object is initialized with the value of the original variable as if there was an implied declaration within the statement block. Data variables in <i>list</i> are separated by commas.
<i>shared</i> (<i>list</i>)	Declares the scope of the data variables in <i>list</i> to be shared across all threads.
<i>default</i> (<i>shared</i> <i>none</i>)	Defines the default data scope of variables in each thread. Only one default clause can be specified on an omp parallel directive.

Specifying **default(shared)** is equivalent to stating each variable in a **shared(list)** clause.

Specifying **default(none)** requires that each data variable visible to the parallelized statement block must be explicitly listed in a data scope clause, with the exception of those variables that are:

- const-qualified,
- specified in an enclosed data scope attribute clause, or,
- used as a loop control variable referenced only by a corresponding **omp for** or **omp parallel for** directive.

<i>copyin</i> (<i>list</i>)	For each data variable specified in <i>list</i> , the value of the data variable in the master thread is copied to the thread-private copies at the beginning of the parallel region. Data variables in <i>list</i> are separated by commas.
-------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Each data variable specified in the **copyin** clause must be a **threadprivate** variable.

<i>reduction</i> (<i>operator</i> : <i>list</i>)	Performs a reduction on all scalar variables in <i>list</i> using the specified <i>operator</i> . Reduction variables in <i>list</i> are separated by commas.
-------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------

A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back into the original value of the shared reduction variable.

Variables specified in the **reduction** clause:

- must be of a type appropriate to the operator.
- must be shared in the enclosing context.
- must not be const-qualified.
- must not have pointer type.

Notes

When a parallel region is encountered, a logical team of threads is formed. Each thread in the team executes all statements within a parallel region except for work-sharing constructs. Work within work-sharing constructs is distributed among the threads in a team.

Loop iterations must be independent before the loop can be parallelized. An implied barrier exists at the end of a parallelized statement block.

Nested parallel regions are always serialized.

Related References

“Pragmas to Control Parallel Processing” on page 344

“#pragma omp for” on page 358

“#pragma omp parallel for” on page 363

“#pragma omp parallel sections” on page 366

#pragma omp for

Description

The **omp for** directive instructs the compiler to distribute loop iterations within the team of threads that encounters this work-sharing construct.

Syntax

```
#pragma omp for [clause[ clause] ...]  
<for_loop>
```

where *clause* is any of the following:

<code>private (<i>list</i>)</code>	Declares the scope of the data variables in <i>list</i> to be private to each thread. Data variables in <i>list</i> are separated by commas.
<code>firstprivate (<i>list</i>)</code>	Declares the scope of the data variables in <i>list</i> to be private to each thread. Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in <i>list</i> are separated by commas.
<code>lastprivate (<i>list</i>)</code>	Declares the scope of the data variables in <i>list</i> to be private to each thread. The final value of each variable in <i>list</i> , if assigned, will be the value assigned to that variable in the last iteration. Variables not assigned a value will have an indeterminate value. Data variables in <i>list</i> are separated by commas.
<code>reduction (<i>operator</i>:<i>list</i>)</code>	Performs a reduction on all scalar variables in <i>list</i> using the specified <i>operator</i> . Reduction variables in <i>list</i> are separated by commas.

A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back into the original value of the shared reduction variable.

Variables specified in the **reduction** clause:

- must be of a type appropriate to the operator.
- must be shared in the enclosing context.
- must not be const-qualified.
- must not have pointer type.

`ordered` Specify this clause if an ordered construct is present within the dynamic extent of the **omp for** directive.

schedule (*type*)

Specifies how iterations of the **for** loop are divided among available threads. Acceptable values for *type* are:

dynamic

Iterations of a loop are divided into chunks of size $\text{ceiling}(\text{number_of_iterations} / \text{number_of_threads})$.

Chunks are dynamically assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.

dynamic,*n*

As above, except chunks are set to size *n*. *n* must be an integral assignment expression of value 1 or greater.

guided

Chunks are made progressively smaller until the default minimum chunk size is reached. The first chunk is of size

$\text{ceiling}(\text{number_of_iterations} / \text{number_of_threads})$.

Remaining chunks are of size

$\text{ceiling}(\text{number_of_iterations_remaining} / \text{number_of_threads})$.

The minimum chunk size is 1.

Chunks are assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.

guided,*n*

As above, except the minimum chunk size is set to *n*. *n* must be an integral assignment expression of value 1 or greater.

runtime

Scheduling policy is determined at run-time. Use the OMP_SCHEDULE environment variable to set the scheduling type and chunk size.

static

Iterations of a loop are divided into chunks of size $\text{ceiling}(\text{number_of_iterations} / \text{number_of_threads})$. Each thread is assigned a separate chunk.

This scheduling policy is also known as *block scheduling*.

static,*n*

Iterations of a loop are divided into chunks of size *n*. Each chunk is assigned to a thread in *round-robin* fashion.

n must be an integral assignment expression of value 1 or greater.

This scheduling policy is also known as *block cyclic scheduling*.

static,1

Iterations of a loop are divided into chunks of size 1. Each chunk is assigned to a thread in *round-robin* fashion.

This scheduling policy is also known as *cyclic scheduling*.

nowait

Use this clause to avoid the implied **barrier** at the end of the **for** directive. This is useful if you have multiple independent work-sharing sections or iterative loops within a given parallel region. Only one **nowait** clause can appear on a given **for** directive.

and where *for_loop* is a **for** loop construct with the following canonical shape:

```
for (init_expr; exit_cond; incr_expr)  
  statement
```

where:

<i>init_expr</i>	takes form:	<i>iv</i> = <i>b</i> <i>integer-type iv</i> = <i>b</i>
<i>exit_cond</i>	takes form:	<i>iv</i> <= <i>ub</i> <i>iv</i> < <i>ub</i> <i>iv</i> >= <i>ub</i> <i>iv</i> > <i>ub</i>
<i>incr_expr</i>	takes form:	++ <i>iv</i> <i>iv</i> ++ -- <i>iv</i> <i>iv</i> -- <i>iv</i> += <i>incr</i> <i>iv</i> -= <i>incr</i> <i>iv</i> = <i>iv</i> + <i>incr</i> <i>iv</i> = <i>incr</i> + <i>iv</i> <i>iv</i> = <i>iv</i> - <i>incr</i>

and where:

<i>iv</i>	Iteration variable. The iteration variable must be a signed integer not modified anywhere within the for loop. It is implicitly made private for the duration of the for operation. If not specified as lastprivate , the iteration variable will have an indeterminate value after the operation completes..
<i>b, ub, incr</i>	Loop invariant signed integer expressions. No synchronization is performed when evaluating these expressions and evaluated side effects may result in indeterminate values..

Notes

Program sections using the **omp for** pragma must be able to produce a correct result regardless of which thread executes a particular iteration. Similarly, program correctness must not rely on using a particular scheduling algorithm.

The **for** loop iteration variable is implicitly made private in scope for the duration of loop execution. This variable must not be modified within the body of the **for** loop. The value of the increment variable is indeterminate unless the variable is specified as having a data scope of **lastprivate**.

An implicit barrier exists at the end of the **for** loop unless the **nowait** clause is specified.

Restrictions are:

- The **for** loop must be a structured block, and must not be terminated by a **break** statement.
- Values of the loop control expressions must be the same for all iterations of the loop.
- An **omp for** directive can accept only one **schedule** clauses.
- The value of *n* (chunk size) must be the same for all threads of a parallel region.

Related References

"Pragmas to Control Parallel Processing" on page 344

"#pragma omp parallel for" on page 363

#pragma omp ordered

Description

The **omp ordered** directive identifies a structured block of code that must be executed in sequential order.

Syntax

```
#pragma omp ordered  
    statement_block
```

Notes

The **omp ordered** directive must be used as follows:

- It must appear within the extent of a **omp for** or **omp parallel for** construct containing an **ordered** clause.
- It applies to the statement block immediately following it. Statements in that block are executed in the same order in which iterations are executed in a sequential loop.
- An iteration of a loop must not execute the same **omp ordered** directive more than once.
- An iteration of a loop must not execute more than one distinct **omp ordered** directive.

Related References

“Pragmas to Control Parallel Processing” on page 344

“#pragma omp for” on page 358

“#pragma omp parallel for” on page 363

#pragma omp parallel for

Description

The **omp parallel for** directive effectively combines the **omp parallel** and **omp for** directives. This directive lets you define a parallel region containing a single **for** directive in one step.

Syntax

```
#pragma omp parallel for [clause[ clause] ...]  
<for_loop>
```

Notes

All clauses and restrictions described in the **omp parallel** and **omp for** directives apply to the **omp parallel for** directive.

Related References

“Pragmas to Control Parallel Processing” on page 344

“#pragma omp for” on page 358

“#pragma omp parallel” on page 356

#pragma omp section, #pragma omp sections

Description

The **omp sections** directive distributes work among threads bound to a defined parallel region.

Syntax

```
#pragma omp sections [clause[ clause] ...]
{
    [#pragma omp section]
    statement-block
    [#pragma omp section]
    statement-block
    .
    .
    .
}
```

where *clause* is any of the following:

private (<i>list</i>)	Declares the scope of the data variables in <i>list</i> to be private to each thread. Data variables in <i>list</i> are separated by commas.
firstprivate (<i>list</i>)	Declares the scope of the data variables in <i>list</i> to be private to each thread. Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in <i>list</i> are separated by commas.
lastprivate (<i>list</i>)	Declares the scope of the data variables in <i>list</i> to be private to each thread. The final value of each variable in <i>list</i> , if assigned, will be the value assigned to that variable in the last section . Variables not assigned a value will have an indeterminate value. Data variables in <i>list</i> are separated by commas.
reduction (<i>operator</i> : <i>list</i>)	Performs a reduction on all scalar variables in <i>list</i> using the specified <i>operator</i> . Reduction variables in <i>list</i> are separated by commas.

A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back into the original value of the shared reduction variable.

Variables specified in the **reduction** clause:

- must be of a type appropriate to the operator.
- must be shared in the enclosing context.
- must not be const-qualified.
- must not have pointer type.

nowait	Use this clause to avoid the implied barrier at the end of the sections directive. This is useful if you have multiple independent work-sharing sections within a given parallel region. Only one nowait clause can appear on a given sections directive.
--------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Notes

The **omp section** directive is optional for the first program code segment inside the **omp sections** directive. Following segments must be preceded by an **omp section** directive. All **omp section** directives must appear within the lexical construct of the program source code segment associated with the **omp sections** directive.

When program execution reaches a **omp sections** directive, program segments defined by the following **omp section** directive are distributed for parallel

execution among available threads. A barrier is implicitly defined at the end of the larger program region associated with the **omp sections** directive unless the **nowait** clause is specified.

Related References

“Pragmas to Control Parallel Processing” on page 344

“#pragma omp parallel sections” on page 366

#pragma omp parallel sections

Description

The **omp parallel sections** directive effectively combines the **omp parallel** and **omp sections** directives. This directive lets you define a parallel region containing a single **sections** directive in one step.

Syntax

```
#pragma omp parallel sections [clause[ clause] ...]
{
    [#pragma omp section]
    statement-block
    [#pragma omp section]
    statement-block
    .
    .
    .
}
]
```

Notes

All clauses and restrictions described in the **omp parallel** and **omp sections** directives apply to the **omp parallel sections** directive.

Related References

“Pragmas to Control Parallel Processing” on page 344

“#pragma omp parallel” on page 356

“#pragma omp section, #pragma omp sections” on page 364

#pragma omp single

Description

The **omp single** directive identifies a section of code that must be run by a single available thread.

Syntax

```
#pragma omp single [clause[ clause] ...]  
    statement_block
```

where *clause* is any of the following:

private (<i>list</i>)	Declares the scope of the data variables in <i>list</i> to be private to each thread. Data variables in <i>list</i> are separated by commas.
firstprivate (<i>list</i>)	Declares the scope of the data variables in <i>list</i> to be private to each thread. Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in <i>list</i> are separated by commas.
nowait	Use this clause to avoid the implied barrier at the end of the single directive. Only one nowait clause can appear on a given single directive.

Notes

An implied barrier exists at the end of a parallelized statement block unless the **nowait** clause is specified.

Related References

“Pragmas to Control Parallel Processing” on page 344

#pragma omp master

Description

The **omp master** directive identifies a section of code that must be run only by the master thread.

Syntax

```
#pragma omp master  
    statement_block
```

Notes

Threads other than the master thread will not execute the statement block associated with this construct.

No implied barrier exists on either entry to or exit from the master section.

Related References

“Pragmas to Control Parallel Processing” on page 344

#pragma omp critical

Description

The **omp critical** directive identifies a section of code that must be executed by a single thread at a time.

Syntax

```
#pragma omp critical [(name)]  
    statement_block
```

where *name* can optionally be used to identify the critical region. Identifiers naming a critical region have external linkage and occupy a namespace distinct from that used by ordinary identifiers.

Notes

A thread waits at the start of a critical region identified by a given name until no other thread in the program is executing a critical region with that same name. Critical sections not specifically named by **omp critical** directive invocation are mapped to the same unspecified name.

Related References

“Pragmas to Control Parallel Processing” on page 344

#pragma omp barrier

Description

The **omp barrier** directive identifies a synchronization point at which threads in a parallel region will wait until all other threads in that section reach the same point. Statement execution past the **omp barrier** point then continues in parallel.

Syntax

```
#pragma omp barrier
```

Notes

The **omp barrier** directive must appear within a block or compound statement. For example:

```
if (x!=0) {  
    #pragma omp barrier    /* valid usage    */  
}  
if (x!=0)  
    #pragma omp barrier    /* invalid usage */
```

Related References

“Pragmas to Control Parallel Processing” on page 344

#pragma omp flush

Description

The **omp flush** directive identifies a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory.

Syntax

```
#pragma omp flush [ (list) ]
```

where *list* is a comma-separated list of variables that will be synchronized.

Notes

If *list* includes a pointer, the pointer is flushed, not the object being referred to by the pointer. If *list* is not specified, all shared objects are synchronized except those inaccessible with automatic storage duration.

An implied **flush** directive appears in conjunction with the following directives:

- **omp barrier**
- Entry to and exit from **omp critical**.
- Exit from **omp parallel**.
- Exit from **omp for**.
- Exit from **omp sections**.
- Exit from **omp single**.

The **omp flush** directive must appear within a block or compound statement. For example:

```
if (x!=0) {  
    #pragma omp flush    /* valid usage    */  
}  
if (x!=0)  
    #pragma omp flush    /* invalid usage */
```

Related References

“Pragmas to Control Parallel Processing” on page 344

“#pragma omp barrier” on page 370

“#pragma omp critical” on page 369

“#pragma omp for” on page 358

“#pragma omp parallel” on page 356

“#pragma omp parallel for” on page 363

“#pragma omp parallel sections” on page 366

“#pragma omp section, #pragma omp sections” on page 364

“#pragma omp single” on page 367

#pragma omp threadprivate

Description

The **omp threadprivate** directive defines the scope of selected file-scope data variables as being private to a thread, but file-scope visible within that thread.

Syntax

```
#pragma omp threadprivate (list)
```

where *list* is a comma-separated list of variables.

Notes

Each copy of an **omp threadprivate** data variable is initialized once prior to first use of that copy. If an object is changed before being used to initialize a **threadprivate** data variable, behavior is unspecified.

A thread must not reference another thread's copy of an **omp threadprivate** data variable. References will always be to the master thread's copy of the data variable when executing serial and master regions of the program.

Use of the **omp threadprivate** directive is governed by the following points:

- An **omp threadprivate** directive must appear at file scope outside of any definition or declaration.
- A data variable must be declared with file scope prior to inclusion in an **omp threadprivate** directive *list*.
- An **omp threadprivate** directive and its *list* must lexically precede any reference to a data variable found in that *list*.
- A data variable specified in an **omp threadprivate** directive in one translation unit must also be specified as such in all other translation units in which it is declared.
- Data variables specified in an **omp threadprivate** *list* must not appear in any clause other than the **copyin**, **schedule**, and **if** clauses.
- The address of a data variable in an **omp threadprivate** *list* is not an address constant.
- A data variable specified in an **omp threadprivate** *list* must not have an incomplete or reference type.

Related References

"Pragmas to Control Parallel Processing" on page 344

Acceptable Compiler Mode and Processor Architecture Combinations

You can use the **-q32**, **-q64**, **-qarch**, and **-qtune** compiler options to optimize the output of the compiler to suit:

- the broadest possible selection of target processors,
- a range of processors within a given processor architecture family,
- a single specific processor.

Generally speaking, the options do the following:

- **-q32** selects 32-bit execution mode.
- **-q64** selects 64-bit execution mode.
- **-qarch** selects the general family processor architecture for which instruction code should be generated. Certain **-qarch** settings produce code that will run *only* on RS/6000 systems that support *all* of the instructions generated by the compiler in response to a chosen **-qarch** setting.
- **-qtune** selects the specific processor for which compiler output is optimized. Some **-qtune** settings can also be specified as **-qarch** options, in which case they do not also need to be specified as a **-qtune** option. The **-qtune** option influences only the performance of the code when running on a particular system but does not determine where the code will run.

There are three main families of RS/6000 machines:

- POWER
- POWER2
- PowerPC

All RS/6000 machines share a common set of instructions, but may also include additional instructions unique to a given processor or processor family.

For example, the POWER2 instruction set is a superset of the POWER instructions set. The PowerPC instruction set includes some instructions not available on POWER systems but does not support all of the POWER instruction set. It also includes a number of POWER2 instructions not available in the POWER instruction set. Also, some features found in the POWER2 instruction set may or may not be implemented on particular PowerPC processors. These optional feature groups include:

- support for the graphics instruction group
- support for the sqrt instruction group
- support for 64-bit mode (**-q64** compiler option)

Other differences in processor features, which may affect instruction code produced by the compiler, are shown below:

Processor	graphics support	sqrt support	64-bit support	large page support
601	no	no	no	no
603	yes	no	no	no
604	yes	no	no	no
rs64a	no	no	yes	no
rs64b	yes	yes	yes	no
rs64c	yes	yes	yes	no
pwr3	yes	yes	yes	no
pwr4	yes	yes	yes	yes, with AIX v5.1D or later

If you want to generate code that will run across a variety of processors, use the following guidelines to select the appropriate **-qarch** and/or **-qtune** compiler options. Code compiled with:

- **-qarch=com** will run on any RS/6000.
- **-qarch=pwr** will run on any POWER or POWER2 machine.
- **-qarch=pwr2** (or **pwr2s**, **pwrx**, **p2sc**) will run only on POWER2 machines.
- **-qarch=pwr4** (or equivalent option **-qarch=gp**) will run only on Power4 machines.
- **-qarch=ppc** will run only on all PowerPC machines.
- **-q64** will run only on PowerPC machines with 64-bit support
- other **-qarch** options that refer to specific processors will run on any functionally equivalent PowerPC machine. In the examples found in the table below, code compiled with **-qarch=pwr3** will also run on a **rs64b** but not on a **rs64a**. Similarly, code compiled with **-qarch=603** will run on a **pwr3** but not on a **rs64a**.

If you want to generate code optimized specifically for a particular processor, acceptable combinations of **-q32**, **-q64**, **-qarch**, and **-qtune** compiler options are shown in the following tables (one table each for 32-bit and 64-bit execution modes). If you specify incompatible combinations of these options, the compiler will assume its own option selections, as described in (Specify Compiler Options for Architecture-Specific, 32- or 64-bit Compilation).

The default execution mode is 32-bit unless the **OBJECT_MODE** environment variable is set to 64.

Related Tasks

“Specify Compiler Options for Architecture-Specific, 32- or 64-bit Compilation” on page 29

“Set Environment Variables to Select 64- or 32-bit Modes” on page 20

Related References

“Compiler Command Line Options” on page 61

“32, 64” on page 73

“arch” on page 83

“tune” on page 277

Acceptable -qarch/-qtune Combinations for 32-bit Execution Mode				
-qarch option	Predefined Macro(s)	Default -qtune settings	Available -qtune setting(s)	
com	_ARCH_COM	pwr2	pwr pwr2 pwr2s pwr3 pwr4 pwrx p2sc 601 602 603 604 403 rs64a rs64b rs64c	
pwr	_ARCH_PWR	pwr2	pwr pwr2 pwr2s pwrx p2sc 601	
	_ARCH_PWR_ARCH_PWR2	pwr2	pwr2 pwr2s pwrx p2sc	
	pwr2s	pwr2s	pwr2s	
	p2sc	p2sc	p2sc	
	601	601	601	
	ppc	_ARCH_PPC	604	601 602 603 604 403 rs64a rs64b rs64c pwr3 pwr4
	601	_ARCH_PPC_ARCH_601	601	601
	602	_ARCH_PPC_ARCH_602	602	602
	403	_ARCH_PPC_ARCH_403	403	403
	ppcgr	_ARCH_PPC_ARCH_PPCGR	604	603 604
ppcgrsq	_ARCH_PPC_ARCH_PPCGR_ARCH_603	603	603	
	_ARCH_PPC_ARCH_PPCGR_ARCH_604	604	604	
	pwr3	604	603 604	
	pwr4	pwr3	pwr3	
	rs64b	pwr4	pwr4	
	rs64c	rs64b	rs64b	
	rs64a	rs64c	rs64c	
		_ARCH_PPC_ARCH_PPCGRSQ	rs64a	rs64a
		_ARCH_PPC_ARCH_PPCGRSQ_ARCH_PPCGRSQ_ARCH_PWR3		
		_ARCH_PPC_ARCH_PPCGR_ARCH_PPCGRSQ_ARCH_PWR4		
	_ARCH_PPC_ARCH_PPCGR_ARCH_PPCGRSQ_ARCH_RS64B			
	_ARCH_PPC_ARCH_PPCGR_ARCH_PPCGRSQ_ARCH_RS64C			

Acceptable -qarch/-qtune Combinations for 64-bit Execution Mode			
-qarch option	Predefined Macro(s)	Default -qtune setting	Available -qtune setting(s)
com	_ARCH_COM	pwr3	auto pwr3 pwr4 rs64a rs64b rs64c
ppc	_ARCH_PPC	pwr3	auto rs64a rs64b rs64c pwr3 pwr4
	_ARCH_PPC_ARCH_PPCGR	pwr3	auto pwr3
	ppcgr	pwr3	auto pwr3
	ppcgrsq	_ARCH_PPC_ARCH_PPCGR_ARCH_PPCGRSQ	auto pwr3
		_ARCH_PPC_ARCH_PPCGR_ARCH_PPCGRSQ_ARCH_PWR3	pwr3
	_ARCH_PPC_ARCH_PPCGR_ARCH_PPCGRSQ_ARCH_PWR4	pwr4	auto pwr4
	_ARCH_PPC_ARCH_PPCGR_ARCH_PPCGRSQ_ARCH_RS64B	rs64b	auto rs64b
	_ARCH_PPC_ARCH_PPCGR_ARCH_PPCGRSQ_ARCH_RS64C	rs64c	auto rs64c
	_ARCH_PPC_ARCH_RS64A	rs64a	auto rs64a

Related Tasks

“Specify Compiler Options for Architecture-Specific, 32- or 64-bit Compilation” on page 29

“Set Environment Variables to Select 64- or 32-bit Modes” on page 20

Related References

“Compiler Command Line Options” on page 61

“32, 64” on page 73

“arch” on page 83

“tune” on page 277


Compiler Messages

This section outlines some of the basic reporting mechanisms the compiler uses to describe compilation errors.

- “Message Severity Levels and Compiler Response”
- “Compiler Return Codes”
- “Compiler Message Format” on page 380

Message Severity Levels and Compiler Response

The following table shows the compiler response associated with each level of message severity.

Letter	Severity	Compiler Response
I	Informational	Compilation continues. The message reports conditions found during compilation.
W	Warning	Compilation continues. The message reports valid, but possibly unintended, conditions.
E	Error	 Compilation continues and object code is generated. Error conditions exist that the compiler can correct, but the program might not run correctly.
S	Severe error	Compilation continues, but object code is not generated. Error conditions exist that the compiler cannot correct.
U	Unrecoverable error	The compiler halts. An internal compiler error has been found. This message should be reported to your IBM service representative.

Related Concepts

“Compiler Message and Listing Information” on page 8

Related References

“Compiler Return Codes”

“Compiler Message Format” on page 380

“halt” on page 143

“maxerr” on page 206

“haltonmsg” on page 144

Compiler Return Codes

At the end of compilation, the compiler sets the return code to zero under any of the following conditions:

- No messages are issued.
- The highest severity level of all errors diagnosed is less than the setting of the **-qhalt** compiler option, and the number of errors did not reach the limit set by the **-qmaxerr** compiler option.
- No message specified by the **-qhaltonmsg** compiler option is issued.

Otherwise, the compiler sets the return code to one of the following values:

Return Code	Error Type
1	Any error with a severity level higher than the setting of the halt compiler option has been detected.
40	An option error or an unrecoverable error has been detected.
41	A configuration file error has been detected.
250	An out-of-memory error has been detected. The xlCcommand cannot allocate any more memory for its use.
251	A signal-received error has been detected. That is, an unrecoverable error or interrupt signal has occurred.
252	A file-not-found error has been detected.
253	An input/output error has been detected: files cannot be read or written to.
254	A fork error has been detected. A new process cannot be created.
255	An error has been detected while the process was running.

Note: Errors may also occur at runtime. For example, a runtime return code of 99 indicates that a static initialization has failed.

Related Concepts

"Compiler Message and Listing Information" on page 8

Related References

"Message Severity Levels and Compiler Response" on page 379

"Compiler Message Format"

"halt" on page 143

"maxerr" on page 206

"haltonmsg" on page 144

Compiler Message Format

Diagnostic messages have the following format when the **-qnosrcmsg** option is active (which is the default):

"file", line line_number.column_number: 15dd-nnn (severity) text.

where:

file is the name of the C or C++ source file with the error.
line_number is the line number of the error.
column_number is the column number for the error
15 is the compiler product identifier

<i>cc</i>	is a two-digit code indicating the VisualAge C++ component that issued the message. <i>cc</i> can have the following values:
00	- code generating or optimizing message
01	- compiler services message.
05	- message specific to the C compiler
06	- message specific to the C compiler
40	- message specific to the C++ compiler
47	- message specific to munch utility
86	- message specific to interprocedural analysis (IPA).
<i>nnn</i>	is the message number
<i>severity</i>	is a letter representing the severity of the error
<i>text</i>	is a message describing the error

Diagnostic messages have the following format when the **-qsrcmsg** option is specified:

x - 15dd-nnn(severity) text.

where *x* is a letter referring to a finger in the finger line.

Related Concepts

"Compiler Message and Listing Information" on page 8

Related References

"Message Severity Levels and Compiler Response" on page 379

"Compiler Return Codes" on page 379

"halt" on page 143

"maxerr" on page 206

"haltonmsg" on page 144

Parallel Processing Support

This section contains information on environment variables and built-in functions used to control parallel processing. Topics in this section are:

- “IBM SMP Run-time Options for Parallel Processing”
- “OpenMP Run-time Options for Parallel Processing” on page 386
- “Built-in Functions Used for Parallel Processing” on page 388

IBM SMP Run-time Options for Parallel Processing

Run-time options affecting SMP parallel processing can be specified with the XLSMPOPTS environment variable. This environment variable must be set before you run an application, and uses basic syntax of the form:



Parallelization run-time options can also be specified using OMP environment variables. When runtime options specified by OMP- and XLSMPOPTS-specific environment variables conflict, OMP options will prevail.

Note: You must use thread-safe compiler mode invocations when compiling parallelized program code.

SMP run-time option settings for the XLSMPOPTS environment variable are shown below, grouped by category:

Scheduling Algorithm Options

**XLSMPOPTS
Environment Variable
Option**

`schedule=algorithm=[n]`

Description

This option specifies the scheduling algorithm used for loops not explicitly assigned a scheduling algorithm with the **ibm schedule** pragma.

Valid options for *algorithm* are:

- guided
- affinity
- dynamic
- static

If specified, the value of *n* must be an integer value of 1 or greater.

The default is scheduling algorithm is **static**.

See **#pragma ibm schedule** for a description of these algorithms.

Parallel Environment Options

XLSPMPOPTS Environment Variable Option	Description
<code>parthds=num</code>	<p><i>num</i> represents the number of parallel threads requested, which is usually equivalent to the number of processors available on the system.</p> <p>Some applications cannot use more threads than the maximum number of processors available. Other applications can experience significant performance improvements if they use more threads than there are processors. This option gives you full control over the number of user threads used to run your program.</p> <p>The default value for <i>num</i> is the number of processors available on the system.</p>
<code>usrthds=num</code>	<p><i>num</i> represents the number of user threads expected.</p> <p>This option should be used if the program code explicitly creates threads, in which case <i>num</i> should be set to the number of threads created.</p> <p>The default value for <i>num</i> is 0.</p>
<code>stack=num</code>	<p><i>num</i> specifies the largest amount of space required for a thread's stack.</p> <p>The default value for <i>num</i> is 4194304.</p>

Performance Tuning Options

XLSPMPOPTS Environment Variable Option	Description
<code>spins=num</code>	<p><i>num</i> represents the number of loop spins before a yield occurs.</p> <p>When a thread completes its work, the thread continues executing in a tight loop looking for new work. One complete scan of the work queue is done during each busy-wait state. An extended busy-wait state can make a particular application highly responsive, but can also harm the overall responsiveness of the system unless the thread is given instructions to periodically scan for and yield to requests from other applications.</p> <p>A complete busy-wait state for benchmarking purposes can be forced by setting both spins and yields to 0.</p> <p>The default value for <i>num</i> is 100.</p>
<code>yields=num</code>	<p><i>num</i> represents the number of yields before a sleep occurs.</p> <p>When a thread sleeps, it completely suspends execution until another thread signals that there is work to do. This provides better system utilization, but also adds extra system overhead for the application.</p> <p>The default value for <i>num</i> is 100.</p>

XLSMPOPTS Environment Variable Option	Description
delays= <i>num</i>	<p><i>num</i> represents a period of do-nothing delay time between each scan of the work queue. Each unit of delay is achieved by running a single no-memory-access delay loop.</p> <p>The default value for <i>num</i> is 500.</p>

Dynamic Profiling Options

XLSMPOPTS Environment Variable Option	Description
profilefreq= <i>num</i>	<p><i>num</i> represents the sampling rate at which each loop is revisited to determine appropriateness for parallel processing.</p> <p>The run-time library uses dynamic profiling to dynamically tune the performance of automatically-parallelized loops. Dynamic profiling gathers information about loop running times to determine if the loop should be run sequentially or in parallel the next time through. Threshold running times are set by the parthreshold and seqthreshold dynamic profiling options, described below.</p> <p>If <i>num</i> is 0, all profiling is turned off, and overheads that occur because of profiling will not occur. If <i>num</i> is greater than 0, running time of the loop is monitored once every <i>num</i> times through the loop.</p> <p>The default for <i>num</i> is 16. The maximum sampling rate is 32. Higher values of <i>num</i> are changed to 32.</p>
parthreshold= <i>mSec</i>	<p><i>mSec</i> specifies the expected running time in milliseconds below which a loop must be run sequentially. <i>mSec</i> can be specified using decimal places.</p> <p>If parthreshold is set to 0, a parallelized loop will never be serialized by the dynamic profiler.</p> <p>The default value for <i>mSec</i> is 0.2 milliseconds.</p>
seqthreshold= <i>mSec</i>	<p><i>mSec</i> specifies the expected running time in milliseconds beyond which a loop that has been serialized by the dynamic profiler must revert to being run in parallel mode again. <i>mSec</i> can be specified using decimal places.</p> <p>The default value for <i>mSec</i> is 5 milliseconds.</p>

Related Concepts

“Program Parallelization” on page 9
 “IBM SMP Directives” on page 9
 “OpenMP Directives” on page 10

Related References

“OpenMP Run-time Options for Parallel Processing” on page 386
 “smp” on page 252
 “Pragmas to Control Parallel Processing” on page 344
 “Built-in Functions Used for Parallel Processing” on page 388

For complete information about the OpenMP Specification, see:
OpenMP Web site
OpenMP Specification.

OpenMP Run-time Options for Parallel Processing

OpenMP run-time options affecting parallel processing are set by specifying OpenMP environment variables. These environment variables, which must be set before you run an application, use syntax of the form:

►►—*env_variable*—=*option_and_args*—►►

Parallelization run-time options can also be specified by the XLSMPOPTS environment variable. When OMP and XLSMPOPTS run-time options conflict, OMP options will prevail.

Note: You must use thread-safe compiler mode invocations when compiling parallelized program code.

OpenMP run-time options fall into different categories as described below:

Scheduling Algorithm Environment Variable

OMP_SCHEDULE=*algorithm*

This option specifies the scheduling algorithm used for loops not explicitly assigned a scheduling algorithm with the **omp schedule** directive. For example:

```
OMP_SCHEDULE="guided, 4"
```

Valid options for *algorithm* are:

- dynamic[, *n*]
- guided[, *n*]
- runtime
- static[, *n*]

If specified, the value of *n* must be an integer value of 1 or greater.

The default scheduling algorithm is **static**.

See “Scheduling Algorithm Options” on page 383 for a description of these algorithms.

Parallel Environment Environment Variables

`OMP_NUM_THREADS=num` *num* represents the number of parallel threads requested, which is usually equivalent to the number of processors available on the system.

This number can be overridden during program execution by calling the `omp_set_num_threads()` runtime library function.

Some applications cannot use more threads than the maximum number of processors available. Other applications can experience significant performance improvements if they use more threads than there are processors. This option gives you full control over the number of user threads used to run your program.

The default value for *num* is the number of processors available on the system.

`OMP_NESTED=TRUE|FALSE`

This environment variable enables or disables nested parallelism. The setting of this environment variable can be overridden by calling the `omp_set_nested()` runtime library function.

If nested parallelism is disabled, nested parallel regions are serialized and run in the current thread.

In the current implementation, nested parallel regions are always serialized. As a result, `OMP_SET_NESTED` does not have any effect, and `omp_get_nested()` always returns 0. If `-qsmp=nested_par` option is on (only in non-strict OMP mode), nested parallel regions may employ additional threads as available. However, no new team will be created to run nested parallel regions.

The default value for `OMP_NESTED` is `FALSE`.

Dynamic Profiling Environment Variable

`OMP_DYNAMIC=TRUE|FALSE` This environment variable enables or disables dynamic adjustment of the number of threads available for running parallel regions.

If set to `TRUE`, the number of threads available for executing parallel regions may be adjusted at runtime to make the best use of system resources. See the description for `profilefreq=num` in “Dynamic Profiling Options” on page 385 for more information.

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

The default setting is `TRUE`.

Related Concepts

“Program Parallelization” on page 9

“IBM SMP Directives” on page 9

“OpenMP Directives” on page 10

Related References

“IBM SMP Run-time Options for Parallel Processing” on page 383

“smp” on page 252

“Pragmas to Control Parallel Processing” on page 344

“Built-in Functions Used for Parallel Processing”

For complete information about the OpenMP Specification, see:

OpenMP Web site

OpenMP Specification.

Built-in Functions Used for Parallel Processing

Use these built-in functions to obtain information about the parallel environment. Function definitions for the **omp_** functions can be found in the **omp.h** header file.

Function Prototype	Description
<code>int __parthds(void)</code>	This function returns the value of the parthds run-time option. If the parthds option is not explicitly set by the user, the function returns the default value set by the run-time library. If the -qsmp compiler option was not specified during program compilation, this function returns 1 regardless of run-time options selected.
<code>int __usrthds(void)</code>	This function returns the value of the usrthds run-time option. If the usrthds option is not explicitly set by the user, or the -qsmp compiler option was not specified during program compilation, this function returns 0 regardless of run-time options selected.
<code>int omp_get_num_threads(void);</code>	This function returns the number of threads currently in the team executing the parallel region from which it is called.
<code>int omp_get_max_threads(void);</code>	This function returns the maximum value that can be returned by calls to <code>omp_get_num_threads</code> .
<code>int omp_get_thread_num(void);</code>	This function returns the thread number, within its team, of the thread executing the function. The thread number lies between 0 and <code>omp_get_num_threads()-1</code> , inclusive. The master thread of the team is thread 0.
<code>int omp_get_num_procs(void);</code>	This function returns the maximum number of processors that could be assigned to the program.
<code>int omp_in_parallel(void);</code>	This function returns non-zero if it is called within the dynamic extent of a parallel region executing in parallel; otherwise, it returns 0.
<code>void omp_set_dynamic(int dynamic_threads);</code>	This function enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.
<code>int omp_get_dynamic(void);</code>	This function returns non-zero if dynamic thread adjustments enabled and returns 0 otherwise.
<code>void omp_set_nested(int nested);</code>	This function enables or disables nested parallelism.

Function Prototype	Description
int omp_get_nested(void);	This function returns non-zero if nested parallelism is enabled and 0 if it is disabled.
void omp_init_lock(omp_lock_t *lock); void omp_init_nest_lock(omp_nest_lock_t *lock);	These functions provide the only means of initializing a lock. Each function initializes the lock associated with the parameter lock for use in subsequent calls.
void omp_destroy_lock(omp_lock_t *lock); void omp_destroy_nest_lock(omp_nest_lock_t *lock);	These functions ensure that the pointed to lock variable lock is uninitialized.
void omp_set_lock(omp_lock_t *lock); void omp_set_nest_lock(omp_nest_lock_t *lock);	Each of these functions blocks the thread executing the function until the specified lock is available and then sets the lock. A simple lock is available if it is unlocked. A nestable lock is available if it is unlocked or if it is already owned by the thread executing the function.
void omp_unset_lock(omp_lock_t *lock); void omp_unset_nest_lock(omp_nest_lock_t *lock);	These functions provide the means of releasing ownership of a lock.
int omp_test_lock(omp_lock_t *lock); int omp_test_nest_lock(omp_nest_lock_t *lock);	These functions attempt to set a lock but do not block execution of the thread.

Note: In the current implementation, nested parallel regions are always serialized. As a result, **omp_set_nested** does not have any effect, and **omp_get_nested** always returns 0.

For complete information about OpenMP runtime library functions, refer to the OpenMP C/C++ Application Program Interface specification.

Related Concepts

“Program Parallelization” on page 9

Related Tasks

“Set Parallel Processing Run-time Options” on page 20

“Control Parallel Processing with Pragmas” on page 45

Related References

“Pragmas to Control Parallel Processing” on page 344

“smp” on page 252

“IBM SMP Run-time Options for Parallel Processing” on page 383

“OpenMP Run-time Options for Parallel Processing” on page 386

“General Purpose Built-in Functions” on page 393

“LIBANSI Built-in Functions” on page 394

“Use the Subroutine Linkage Conventions in Interlanguage Calls” on page 49

Part 4. Appendixes

Appendix A. Built-in Functions

The compiler provides you with a selection of built-in functions to help you write more efficient programs. This section summarizes the various built-in functions available to you.

- “General Purpose Built-in Functions”
- “LIBANSI Built-in Functions” on page 394
- “Built-in Functions for PowerPC Processors” on page 394

You can also find additional built-in functions to support parallel processing program execution described at “Built-in Functions Used for Parallel Processing” on page 388.

General Purpose Built-in Functions

Name	C/C++ Prototype	Description
__cntlz4	unsigned int __cntlz4(unsigned int);	Count Leading Zeros, 4-Byte Integer
__cntlz8	unsigned int __cntlz8(unsigned long long);	Count Leading Zeros, 8-Byte Integer
__cnttz4	unsigned int __cnttz4(unsigned int);	Count Trailing Zeros, 4-Byte Integer
__cnttz8	unsigned int __cnttz8(unsigned long long);	Count Trailing Zeros, 8-Byte Integer
__fmsub	double __fmsub(double, double, double);	floating point long multiply then sub
__fnabs	double __fnabs(double);	floating point long negative absolute
__fnabss	float __fnabss(float);	floating point short negative absolute
__fnadd	double __fnmadd(double, double, double);	floating point long negative multiply then add
__fnmsub	double __fnmsub(double, double, double);	floating point long negative multiply then sub
__iospace_eieio	(equivalent to: void __iospace_eieio(void);)	I/O Sync Point
__load2r	unsigned short __load2r(unsigned short*);	load 2 byte register
__load4r	unsigned int __load4r(unsigned int*);	load 4 byte register
__prefetch_by_load	void __prefetch_by_load(const void*);	touch a memory location via explicit load
__readflm	double d __readflm();	read floating point status/control register
__setflm	double __setflm(double);	Set Floating Point Status/Control Register
__settrnd	double __setrnd(int);	Set Rounding Mode
__trap	void __trap(int);	trap

Related References

“General Purpose Built-in Functions” on page 393

“LIBANSI Built-in Functions”

“Built-in Functions for PowerPC Processors”

“Built-in Functions Used for Parallel Processing” on page 388

LIBANSI Built-in Functions

Name	C/C++ Prototype	Description
<code>__abs</code>	<code>int __abs(int);</code>	integer absolute value
<code>__acos</code>	<code>double __acos(double);</code>	arc-cosine
<code>__alloca</code>	<code>void* __alloca(size_t);</code>	memory allocation on stack
<code>__bcopy</code>	<code>void __bcopy(char*, char*, int);</code>	
<code>__fabs</code>	<code>double __fabs(double);</code>	long float point absolute value
<code>__fabss</code>	<code>float __fabss(float);</code>	short floating point absolute value
<code>__labs</code>	<code>long __labs(long);</code>	long int absolute value
<code>__llabs</code>	<code>long long __llabs(long long);</code>	long long absolute
<code>__memchr</code>	<code>void* __memchr(const void*, int, size_t);</code>	memory character
<code>__memcmp</code>	<code>int __memcmp(const void*, const void*, size_t);</code>	memory compare
<code>__strchr</code>	<code>char* __strchr(const char*, int);</code>	string char
<code>__strcmp</code>	<code>int __strcmp(const char*, const char*);</code>	string compare
<code>__strlen</code>	<code>size_t __strlen(const char*);</code>	string length
<code>__strncmp</code>	<code>int __strncmp(const char*, const char*, size_t);</code>	string numbered compare
<code>__strrchr</code>	<code>char* __strrchr(const char*, int);</code>	string reverse char

Related References

“General Purpose Built-in Functions” on page 393

“LIBANSI Built-in Functions”

“Built-in Functions for PowerPC Processors”

“Built-in Functions Used for Parallel Processing” on page 388

Built-in Functions for PowerPC Processors

PowerPC platforms support RS/6000 machine instructions not available on other platforms. If performance is critical to your application, the VisualAge C++ compiler provides a set of built-in functions that directly map to certain PowerPC instructions. By using these functions, function call return costs, parameter passing, stack adjustment and all the additional costs related with function invocations are eliminated.

Not all functions described below are supported by all RS/6000 processors. Using an unsupported function will result in an error message being displayed.

Name	Prototype	Return Value or Action Performed
__check_lock_mp	unsigned int __check_lock_mp (const int* <i>addr</i> , int <i>old_value</i> , int <i>new_value</i>)	Check Lock on MultiProcessor systems. Conditionally updates a single word variable atomically. <i>addr</i> specifies the address of the single word variable. <i>old_value</i> specifies the old value to be checked against the value of the single word variable. <i>new_value</i> specifies the new value to be conditionally assigned to the single word variable. The word variable must be aligned on a full word boundary Return values: 1. A return value of false indicates that the single word variable was equal to the old value and has been set to the new value. 2. A return value of true indicates that the single word variable was not equal to the old value and has been left unchanged.
__check_lock_up	unsigned int __check_lock_up (const int* <i>addr</i> , int <i>old_value</i> , int <i>new_value</i>)	Check Lock on UniProcessor systems. Conditionally updates a single word variable atomically. <i>addr</i> specifies the address of the single word variable. <i>old_value</i> specifies the old value to be checked against the value of the single word variable. <i>new_value</i> specifies the new value to be conditionally assigned to the single word variable. The word variable must be aligned on a full word boundary. Return values: <ul style="list-style-type: none"> • A return value of false indicates that the single word variable was equal to the old value, and has been set to the new value. • A return value of true indicates that the single word variable was not equal to the old value and has been left unchanged.
__clear_lock_mp	void __clear_lock_mp (const int* <i>addr</i> , int <i>value</i>)	Clear Lock on MultiProcessor systems. Atomic store of the <i>value</i> into the single word variable at the address <i>addr</i> . The word variable must be aligned on a full word boundary.
__clear_lock_up	void __clear_lock_up (const int* <i>addr</i> , int <i>value</i>)	Clear Lock on UniProcessor systems. Atomic store of the <i>value</i> into the single word variable at the address <i>addr</i> . The word variable must be aligned on a full word boundary.
__dcbt ()	void __dcbt (void *);	Data Cache Block Touch. Loads the block of memory containing the specified address into the data cache.

Name	Prototype	Return Value or Action Performed
<code>__dcbz()</code>	<code>void __dcbz (void *);</code>	Data Cache Block set to Zero. Sets the specified address in the data cache to zero (0).
<code>__eieio</code>	(Compiler will recognize <code>__eieio</code> built-in.)	Extra name for the existing <code>__iospace_eieio</code> built-in. Compiler will recognize <code>__eieio</code> built-in. Everything except for the name is exactly same as for <code>__iospace_eieio</code> . <code>__eieio</code> is consistent with the corresponding PowerPC instruction name.
<code>__fabs()</code>	<code>double __fabs (double);</code>	<code>__fabs (x) = x </code>
<code>__fabss()</code>	<code>float __fabss (float);</code>	<code>__fabss (x) = x </code>
<code>__fcfid</code>	<code>double __fcfid (double)</code>	Floating Convert From Integer Doubleword. The 64bit signed fixedpoint operand is converted to a double-precision floating-point.
<code>__fctid</code>	<code>double __fctid (double)</code>	Floating Convert to Integer Doubleword. The floating-point operand is converted into 64-bit signed fixed-point integer, using the rounding mode specified by <code>FPSCR_{RN}</code> (Floating-Point Rounding Control field in the Floating-Point Status and Control Register).
<code>__fctidz</code>	<code>double __fctidz (double)</code>	Floating Convert to Integer Doubleword with Rounding towards Zero. The floating-point operand is converted into 64-bit signed fixed-point integer, using the rounding mode Round toward Zero
<code>__fctiw</code>	<code>double __fctiw (double)</code>	Floating Convert To Integer Word. The floating-point operand is converted to a 32-bit signed fixed-point integer, using the rounding mode specified by <code>FPSCR_{RN}</code> (Floating-Point Rounding Control field in the Floating-Point Status and Control Register).
<code>__fctiwz</code>	<code>double __fctiwz (double)</code>	Floating Convert To Integer Word with Rounding towards Zero. The floating-point operand is converted to a 32-bit signed fixed-point integer, using the rounding mode Round toward Zero
<code>__fmadd()</code>	<code>double __fmadd (double, double, double);</code>	<code>__fmadd (a, x, y) = [a * x + y]</code>
<code>__fmadds()</code>	<code>float __fmadds (float, float, float);</code>	<code>__fmadds (a, x, y) = [a * x + y]</code>
<code>__fmsubs()</code>	<code>float __fmsubs (float, float, float);</code>	<code>__fmsubs (a, x, y) = [a * x - y]</code>
<code>__fmsub()</code>	<code>double __fmsub (double, double, double);</code>	<code>__fmsub (a, x, y) = [a * x - y]</code>

Name	Prototype	Return Value or Action Performed
<code>__fnabss()</code>	<code>float __fnabss (float);</code>	<code>__fnabss (x) = - x </code>
<code>__fnabs()</code>	<code>double __fnabs (double);</code>	<code>__fnabs (x) = - x </code>
<code>__fnmadd()</code>	<code>double __fnmadd (double, double, double);</code>	<code>__fnmadd (a, x, y) = [- (a * x + y)]</code>
<code>__fnmadds()</code>	<code>float __fnmadds (float, float, float);</code>	<code>__fnmadds (a, x, y) = [- (a * x + y)]</code>
<code>__fnmsub()</code>	<code>double __fnmsub (double double, double);</code>	<code>__fnmsub (a, x, y) = [- (a * x - y)]</code>
<code>__fnmsubs()</code>	<code>float __fnmsubs (float, float, float);</code>	<code>__fnmsubs (a, x, y) = [- (a * x - y)]</code>
<code>__fres()</code>	<code>float __fres (float);</code>	<code>__fres (x) = [(estimate of) 1.0/x]</code>
<code>__fsqrt()</code>	<code>double __fsqrt (double);</code>	<code>__fsqrt (x) = square root of x</code>
<code>__fsqrts()</code>	<code>float __fsqrts (float);</code>	<code>__fsqrts (x) = square root of x</code>
<code>__frsrqte()</code>	<code>double __frsrqte (double);</code>	<code>__frsrqte (x) = [(estimate of) 1.0/sqrt(x)]</code>
<code>__fsel()</code>	<code>double __fsel (double, double, double);</code>	if $(a \geq 0.0)$ then <code>__fsel (a, x, y) = x;</code> else <code>__fsel (a, x, y) = y</code>
<code>__fsels()</code>	<code>float __fsels (float, float, float);</code>	if $(a \geq 0.0)$ then <code>__fsels (a, x, y) = x;</code> else <code>__fsels (a, x, y) = y</code>
<code>__lwsync</code>	(Compiler will recognize <code>__lwsync</code> built-in.)	Extra name for the existing <code>__iospace_lwsync</code> built-in. Compiler will recognize <code>__lwsync</code> built-in. Everything except for the name is exactly same as for <code>__iospace_lwsync</code> . <code>__lwsync</code> is consistent with the corresponding PowerPC instruction name.
<code>__mtfsb0</code>	<code>void __mtfsb0(unsigned int bt)</code>	Move to FPSCR Bit 0. Bit <i>bt</i> of the FPSCR is set to 0. <i>bt</i> must be a constant and $0 \leq bt \leq 31$.
<code>__mtfsb1</code>	<code>void __mtfsb1(unsigned int bt)</code>	Move to FPSCR Bit 1. Bit <i>bt</i> of the FPSCR is set to 1. <i>bt</i> must be a constant and $0 \leq bt \leq 31$.
<code>__mtfsf</code>	<code>void __mtfsf(unsigned int flm, unsigned int frb)</code>	Move to FPSCR Fields. The contents of <i>frb</i> are placed into the FPSCR under control of the field mask specified by <i>flm</i> . The field mask <i>flm</i> identifies the 4bit fields of the FPSCR affected. <i>flm</i> must be a constant 8-bit mask.
<code>__mtfsfi</code>	<code>void __mtfsfi(unsigned int bf, unsigned int u)</code>	Move to FPSCR Field Immediate. The value of the <i>u</i> is placed into FPSCR field specified by <i>bf</i> . <i>bf</i> and <i>u</i> must be constants, with $0 \leq bf \leq 7$ and $0 \leq u \leq 15$.
<code>__mulhd</code>	<code>long long int __mulhd(long long int ra, long long int rb)</code>	Multiply High Doubleword Signed. Returns the highorder 64 bits of the 128bit product of the operands <i>ra</i> and <i>rb</i> .

Name	Prototype	Return Value or Action Performed
__mulhdu	unsigned long long int __mulhdu(unsigned long long int <i>ra</i> , unsigned long long int <i>rb</i>)	Multiply High Doubleword Unsigned. Returns the highorder 64 bits of the 128bit product of the operands <i>ra</i> and <i>rb</i> .
__mulhw	int __mulhw(int <i>ra</i> , int <i>rb</i>)	Multiply High Word Signed. Returns the highorder 32 bits of the 64bit product of the operands <i>ra</i> and <i>rb</i> .
__mulhwu	unsigned int __mulhwu(unsigned int <i>ra</i> , unsigned int <i>rb</i>)	Multiply High Word Unsigned. Returns the highorder 32 bits of the 64bit product of the operands <i>ra</i> and <i>rb</i> .
__rdlam	unsigned long long __rdlam(unsigned long long <i>rs</i> , unsigned int <i>shift</i> , unsigned long long <i>mask</i>)	Rotate Double Left and AND with Mask. The contents of <i>rs</i> are rotated left <i>shift</i> bits. The rotated data is ANDed with the mask and returned as a result. <i>mask</i> must be a constant and represent a contiguous bitfield.
__rldimi	unsigned long long __rldimi(unsigned long long <i>rs</i> , unsigned long long <i>is</i> , unsigned int <i>shift</i> , unsigned long long <i>mask</i>)	Rotate Left Doubleword Immediate then Mask Insert. Rotates <i>rs</i> left <i>shift</i> bits then inserts <i>rs</i> into <i>is</i> under bit mask <i>mask</i> . Shift must be a constant and $0 \leq \text{shift} \leq 63$. <i>mask</i> must be a constant and represent a contiguous bitfield.
__rlwimi	unsigned int __rlwimi(unsigned int <i>rs</i> , unsigned int <i>is</i> , unsigned int <i>shift</i> , unsigned int <i>mask</i>)	Rotate Left Word Immediate then Mask Insert. Rotates <i>rs</i> left <i>shift</i> bits then inserts <i>rs</i> into <i>is</i> under bit mask <i>mask</i> . Shift must be a constant and $0 \leq \text{shift} \leq 31$. <i>mask</i> must be a constant and represent a contiguous bitfield.
__rlwnm	unsigned int __rlwnm(unsigned int <i>rs</i> , unsigned int <i>shift</i> , unsigned int <i>mask</i>)	Rotate Left Word then AND with Mask. Rotates <i>rs</i> left <i>shift</i> bits, then ANDs <i>rs</i> with bit mask <i>mask</i> . <i>mask</i> must be a constant and represent a contiguous bitfield.
__rotatel4	unsigned int __rotatel4(unsigned int <i>rs</i> , unsigned int <i>shift</i>)	Rotate Left Word. Rotates <i>rs</i> left <i>shift</i> bits.
__rotatel8	unsigned long long __rotatel8(unsigned long long <i>rs</i> , unsigned long long <i>shift</i>)	Rotate Left Doubleword. Rotates <i>rs</i> left <i>shift</i> bits.
__stfiw	void __stfiw(const int* <i>addr</i> , double <i>value</i>)	Store Floating-Point as Integer Word. The contents of the loworder 32 bits of <i>value</i> are stored, without conversion, into the word in storage addressed by <i>addr</i> .
__sync	(Compiler will recognize __sync built-in.)	Extra name for the existing __iospace_sync built-in. Compiler will recognize __sync built-in. Everything except for the name is exactly same as for __iospace_sync. __sync is consistent with the corresponding PowerPC instruction name.

Name	Prototype	Return Value or Action Performed										
__tdw	void __tdw(long long <i>a</i> , long long <i>b</i> , unsigned int <i>TO</i>)	<p>Trap Doubleword. Operand <i>a</i> is compared with operand <i>b</i>. This comparison results in five conditions which are ANDed with <i>TO</i>, which must be a constant and $0 \leq TO \leq 31$.</p> <p>If the result is not 0 the system trap handler is invoked. These conditions are as follows:</p> <table data-bbox="1006 514 1456 842"> <tr> <td>0</td> <td>Less Than, using signed comparison.</td> </tr> <tr> <td>1</td> <td>Greater Than, using signed comparison.</td> </tr> <tr> <td>3</td> <td>Equal</td> </tr> <tr> <td>4</td> <td>Less Than, using unsigned comparison.</td> </tr> <tr> <td>5</td> <td>Greater Than, using unsigned comparison.</td> </tr> </table>	0	Less Than, using signed comparison.	1	Greater Than, using signed comparison.	3	Equal	4	Less Than, using unsigned comparison.	5	Greater Than, using unsigned comparison.
0	Less Than, using signed comparison.											
1	Greater Than, using signed comparison.											
3	Equal											
4	Less Than, using unsigned comparison.											
5	Greater Than, using unsigned comparison.											
__trap ()	void __trap (int);	Trap if the parameter is not zero.										
__trapd ()	void __trapd (longlong);	Trap if the parameter is not zero.										
__tw	void __tw(int <i>a</i> , int <i>b</i> , unsigned int <i>TO</i>)	<p>Trap Word. Operand <i>a</i> is compared with operand <i>b</i>. This comparison results in five conditions which are ANDed with <i>TO</i>, which must be a constant and $0 \leq TO \leq 31$.</p> <p>If the result is not 0 the system trap handler is invoked. These conditions are as follows:</p> <table data-bbox="1006 1228 1456 1560"> <tr> <td>0</td> <td>Less Than, using signed comparison.</td> </tr> <tr> <td>1</td> <td>Greater Than, using signed comparison.</td> </tr> <tr> <td>3</td> <td>Equal</td> </tr> <tr> <td>4</td> <td>Less Than, using unsigned comparison.</td> </tr> <tr> <td>5</td> <td>Greater Than, using unsigned comparison.</td> </tr> </table>	0	Less Than, using signed comparison.	1	Greater Than, using signed comparison.	3	Equal	4	Less Than, using unsigned comparison.	5	Greater Than, using unsigned comparison.
0	Less Than, using signed comparison.											
1	Greater Than, using signed comparison.											
3	Equal											
4	Less Than, using unsigned comparison.											
5	Greater Than, using unsigned comparison.											

Related References

“General Purpose Built-in Functions” on page 393

“LIBANSI Built-in Functions” on page 394

“Built-in Functions for PowerPC Processors” on page 394

“Built-in Functions Used for Parallel Processing” on page 388

Appendix B. National Languages Support in VisualAge C++

This and related pages summarize the national language support (NLS) specific to IBM VisualAge C++.

For more information, see the following topics in this section:

- “Converting Files Containing Multibyte Data to New Code Pages”
- “Multibyte Character Support”

See also National Language Support in General Programming Concepts: Writing and Debugging Programs.

Converting Files Containing Multibyte Data to New Code Pages

If you have installed new code pages on your system, you can use the AIX **iconv** migration utility to convert files containing multibyte data to use new code pages. This command converts files containing multibyte data from the **IBM-932** code set to the **IBM-euc** code set.

The **iconv** command is described in the *AIX Commands Reference*. Using the NLS code set converters with the **iconv** command is described in “Converters Overview for Programming” in the *AIX General Programming Concepts*.

Related References

Appendix B, “National Languages Support in VisualAge C++”
“Multibyte Character Support”

See also:

iconv command in Commands Reference, Volume 3: i through m:
Converters Overview for Programming section in AIX 5L Version 5.1 General Programming Concepts: Writing and Debugging Programs

Multibyte Character Support

Support for multibyte characters includes support for wide characters. Generally, wide characters are permitted anywhere multibyte characters are, but they are incompatible with multibyte characters in the same string because their bit patterns differ. Wherever permitted, you can mix single-byte and multibyte characters in the same string.

Note: You must specify the **-qmbcs** option to use multibyte characters anywhere in your program.

In the examples that follow, *multibyte_char* represents any string of one or more multibyte characters.

String Literals and Character Constants

Multibyte characters are supported in string literals and character constants. Strings containing multibyte characters are treated in essentially the same way as strings without multibyte characters. Multibyte characters can appear in several contexts:

- Preprocessor directives

- Macro definitions
- The # and ## operators
- The definition of the macro name in the **-D** compiler option

Wide-character strings can be manipulated the same way as single-byte character strings. The system provides equivalent wide-character and single-byte string functions.

The default storage type for all string literals is read-only. The **-qro** option sets the storage type of string literals to read-only, and the **-qnor** option makes string literals writable.

Note: Because a character constant can store only 1 byte, avoid assigning multibyte characters to character constants. Only the last byte of a multibyte character constant is stored. Use a wide-character representation instead. Wide-character string literals and constants must be prefixed by L. For example:

```
wchar_t *a = L"wide_char_string";
wchar_t b = L'c';
```

Preprocessor Directives

The following preprocessor directives permit multibyte-character constants and string literals:

- **#define**
- **#pragma comment**
- **#include**

Macro Definitions

Because string literals and character constants can be part of **#define** statements, multibyte characters are also permitted in both object-like and function-like macro definitions.

Compiler Options

Multibyte characters can appear in the compiler suboptions that take file names as arguments:

- **-I** *key*
- **-o** *file_name*
- **-B** *prefix*
- **-F** *config_file:stanza*
- **-I** *directory*
- **-L** *directory*

The **-Dname=definition** option permits multibyte characters in the definition of the macro name. In the following example, the first definition is a string literal, and the second is a character constant:

```
-DMYMACRO="kpsmultibyte_chardcs"
-DMYMACRO='multibyte_char'
```

The **-qmbcs** compiler option permits both double-byte and multibyte characters. In other respects, it is equivalent to the **-qdbcs** option, but it should be used when multibyte characters appear in the program.

The listings produced by the **-qlist** and **-qsource** options display the date and time for the appropriate international language. Multibyte characters in the file name of the C or C++ source file also appear in the name of the corresponding list file. For example, a C source file called:

```
multibyte_char.c
```

gives a list file called

```
multibyte_char.lst
```

File Names and Comments

Any file name can contain multibyte characters. The file name can be a relative or absolute path name. For example:

```
#include<multibyte_char/mydir/mysource/multibyte_char.h>  
#include "multibyte_char.h"
```

```
xlc /u/myhome/c_programs/kanji_files/multibyte_char.c -omultibyte_char
```

Multibyte characters are also permitted in comments, if you specify the **-qmbcs** compiler option.

Restrictions

- Multibyte characters are not permitted in identifiers.
- Hexadecimal values for multibyte characters must be in the range of the code page being used.
- You cannot mix wide characters and multibyte characters in macro definitions. For example, a macro expansion that concatenates a wide string and a multibyte string is not permitted.
- Assignment between wide characters and multibyte characters is not permitted.
- Concatenating wide character strings and multibyte character strings is not permitted.

Related References

Appendix B, "National Languages Support in VisualAge C++" on page 401

"Converting Files Containing Multibyte Data to New Code Pages" on page 401

"+ (plus sign)" on page 71

"mbsc, dbcs" on page 209

Appendix C. Problem Solving

Topics in this section are:

- “Message Catalog Errors”
- “Correcting Paging Space Errors During Compilation”

Message Catalog Errors

Before the compiler can compile your program, the message catalogs must be installed and the environment variables **LANG** and **NLSPATH** must be set to a language for which the message catalog has been installed.

If you see the following message during compilation, the appropriate message catalog cannot be opened:

```
Error occurred while initializing the message system in
file: message_file
```

where *message_file* is the name of the message catalog that the compiler cannot open. This message is issued in English only.

You should then verify that the message catalogs and the environment variables are in place and correct. If the message catalog or environment variables are not correct, compilation can continue, but all nondiagnostic messages are suppressed and the following message is issued instead:

```
No message text for message_number.
```

where *message_number* is the IBM VisualAge C++ internal message number. This message is issued in English only.

To determine message catalogs which are installed on your system, list all of the file names for the catalogs using the following command:

```
ls /usr/lib/nls/msg/%L/*.cat
```

where %L is the current primary language environment (locale) setting. The default locale is **C**. The locale for United States English is **en_US**.

The default message catalogs in **/usr/vacpp/exe/default_msg** are called when:

- IBM VisualAge C++ cannot find message catalogs for the locale specified by %L.
- The locale has never been changed from the default, **C**.

For more information about the **NLSPATH** and **LANG** environment variables, see your operating system documentation.

Related Tasks

“Set Environment Variables” on page 19

“Set Environment Variables for the Message and Help Files” on page 20

Correcting Paging Space Errors During Compilation

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

1501-229 Compilation ended due to lack of space.

1501-224 fatal error in ../exe/xlCcode: signal 9 received.

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

Killed.

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

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

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

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

Appendix D. ASCII Character Set

VisualAge C++ uses the American National Standard Code for Information Interchange (ASCII) character set as its collating sequence.

The following table lists the standard ASCII characters in ascending numerical order, with their corresponding decimal, octal, and hexadecimal values. It also shows the control characters with **Ctrl-** notation. For example, the carriage return (ASCII symbol **CR**) appears as **Ctrl-M**, which you enter by simultaneously pressing the **Ctrl** key and the **M** key.

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Meaning
0	0	00	Ctrl-@	NUL	null
1	1	01	Ctrl-A	SOH	start of heading
2	2	02	Ctrl-B	STX	start of text
3	3	03	Ctrl-C	ETX	end of text
4	4	04	Ctrl-D	EOT	end of transmission
5	5	05	Ctrl-E	ENQ	enquiry
6	6	06	Ctrl-F	ACK	acknowledge
7	7	07	Ctrl-G	BEL	bell
8	10	08	Ctrl-H	BS	backspace
9	11	09	Ctrl-I	HT	horizontal tab
10	12	0A	Ctrl-J	LF	new line
11	13	0B	Ctrl-K	VT	vertical tab
12	14	0C	Ctrl-L	FF	form feed
13	15	0D	Ctrl-M	CR	carriage return
14	16	0E	Ctrl-N	SO	shift out
15	17	0F	Ctrl-O	SI	shift in
16	20	10	Ctrl-P	DLE	data link escape
17	21	11	Ctrl-Q	DC1	device control 1
18	22	12	Ctrl-R	DC2	device control 2
19	23	13	Ctrl-S	DC3	device control 3
20	24	14	Ctrl-T	DC4	device control 4
21	25	15	Ctrl-U	NAK	negative acknowledge
22	26	16	Ctrl-V	SYN	synchronous idle
23	27	17	Ctrl-W	ETB	end of transmission block
24	30	18	Ctrl-X	CAN	cancel
25	31	19	Ctrl-Y	EM	end of medium
26	32	1A	Ctrl-Z	SUB	substitute
27	33	1B	Ctrl-[ESC	escape
28	34	1C	Ctrl-\	FS	file separator

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Meaning
29	35	1D	Ctrl-]	GS	group separator
30	36	1E	Ctrl-^	RS	record separator
31	37	1F	Ctrl-_	US	unit separator
32	40	20		SP	digit select
33	41	21		!	exclamation point
34	42	22		"	double quotation mark
35	43	23		#	pound sign, number sign
36	44	24		\$	dollar sign
37	45	25		%	percent sign
38	46	26		&	ampersand
39	47	27		'	apostrophe
40	50	28		(left parenthesis
41	51	29)	right parenthesis
42	52	2A		*	asterisk
43	53	2B		+	addition sign
44	54	2C		,	comma
45	55	2D		-	subtraction sign
46	56	2E		.	period
47	57	2F		/	right slash
48	60	30		0	
49	61	31		1	
50	62	32		2	
51	63	33		3	
52	64	34		4	
53	65	35		5	
54	66	36		6	
55	67	37		7	
56	70	38		8	
57	71	39		9	
58	72	3A		:	colon
59	73	3B		;	semicolon
60	74	3C		<	less than
61	75	3D		=	equal
62	76	3E		>	greater than
63	77	3F		?	question mark
64	100	40		@	at sign
65	101	41		A	
66	102	42		B	
67	103	43		C	

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Meaning
68	104	44		D	
69	105	45		E	
70	106	46		F	
71	107	47		G	
72	110	48		H	
73	111	49		I	
74	112	4A		J	
75	113	4B		K	
76	114	4C		L	
77	115	4D		M	
78	116	4E		N	
79	117	4F		O	
80	120	50		P	
81	121	51		Q	
82	122	52		R	
83	123	53		S	
84	124	54		T	
85	125	55		U	
86	126	56		V	
87	127	57		W	
88	130	58		X	
89	131	59		Y	
90	132	5A		Z	
91	133	5B		[left bracket
92	134	5C		\	left slash, backslash
93	135	5D]	right bracket
94	136	5E		^	hat, circumflex, caret
95	137	5F		_	underscore
96	140	60		`	grave accent
97	141	61		a	
98	142	62		b	
99	143	63		c	
100	144	64		d	
101	145	65		e	
102	146	66		f	
103	147	67		g	
104	150	68		h	
105	151	69		i	
106	152	6A		j	
107	153	6B		k	

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Meaning
108	154	6C		l	
109	155	6D		m	
110	156	6E		n	
111	157	6F		o	
112	160	70		p	
113	161	71		q	
114	162	72		r	
115	163	73		s	
116	164	74		t	
117	165	75		u	
118	166	76		v	
119	167	77		w	
120	170	78		x	
121	171	79		y	
122	172	7A		z	
123	173	7B		{	left brace
124	174	7C			logical or, vertical bar
125	175	7D		}	right brace
126	176	7E		~	similar, tilde
127	177	7F		DEL	delete

Notices

Note to U.S. Government Users Restricted Rights -- use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd. Laboratory
B3/KB7/8200/MKM
8200 Warden Avenue
Markham, Ontario L6G 1C7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

Programming Interface Information

Programming interface information is intended to help you create application software using this program.

General-use programming interface allow the customer to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification, and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and Service Marks

The following terms are trademarks of the International Business Machines Corporation in the United States, or other countries, or both:

AIX
IBM
Open Class
POWER
POWER2
PowerPC
RS/6000
VisualAge

Other company, product, and service names, which may be denoted by a double asterisk(**), may be trademarks or service marks of others.

Industry Standards

The following standards are supported:

- The C language is consistent with the International Standard for Information Systems-Programming Language C (ISO/IEC 9899-1999 (E)).
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).



SC09-4959-00

