

# C for AIX User's Guide





# C for AIX User's Guide



**Note!**

Before using this information and the product it supports, be sure to read the general information under "Notices" on page xv.

## September 1999 Edition

This documentation applies to Version 5 Release 0 of the C for AIX compiler and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

If you have comments about this document, address them to:

IBM Canada Ltd. Laboratory  
Information Development  
2G/345/1150/TOR  
1150 Eglinton Avenue East  
North York, Ontario, Canada, M3C 1H7

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, 1999. All rights reserved.**

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



---

# Contents

<b>Notices</b> . . . . .	xv
Trademarks and Service Marks . . . . .	xv
<b>About this Information</b> . . . . .	xvii
Related Reading . . . . .	xvii
IBM Publications . . . . .	xvii
Non-IBM Publications . . . . .	xviii
<b>Chapter 1. Introducing C for AIX.</b> . . . . .	1
<b>Chapter 2. Setting Up the C for AIX Compilation Environment</b> . . . . .	3
Setting Environment Variables to Select 64- or 32-bit Compilation Modes . . . . .	3
Setting Parallel Processing Run-time Options . . . . .	3
Setting Environment Variables for the Message and Help Files . . . . .	3
Setting Environment Variables in bsh, ksh, or sh Shells . . . . .	4
Setting Environment Variables in csh Shell . . . . .	4
<b>Chapter 3. Using the C for AIX Compiler</b> . . . . .	5
Compiler Modes . . . . .	5
Types of Input Files . . . . .	7
Types of Output Files . . . . .	8
Invoking the Compiler . . . . .	8
Invoking the Linkage Editor . . . . .	9
Compiler Options . . . . .	10
Specifying Compiler Options on the Command Line . . . . .	10
Specifying Compiler Options in Your Program Source Files . . . . .	12
Specifying Compiler Options in a Configuration File . . . . .	13
Specifying Compiler Options for Architecture-Specific, 32- or 64-bit Compilation . . . . .	14
Compiler Message and Listing Information . . . . .	18
Compiler Listings . . . . .	19
Message Severity Levels and Compiler Response . . . . .	20
Compiler Return Codes . . . . .	20
Compiler Message Format . . . . .	21
<b>Chapter 4. Advanced Compiler Usage</b> . . . . .	23
Program Optimization with the C for AIX Compiler . . . . .	23
Optimization Techniques Used by the C for AIX Compiler . . . . .	23
Special Handling of Math and String Library Functions . . . . .	25
Floating Point Operations with the C for AIX Compiler . . . . .	25
RISC System/6000 Floating Point Hardware . . . . .	25
Compile-Time Floating-Point Arithmetic . . . . .	26
Floating-Point Compiler Options . . . . .	27
Rounding Mode Restrictions . . . . .	35
Creating and Using Precompiled Headers . . . . .	35
Minimizing the Size of Object Files . . . . .	36
<b>Chapter 5. Program Parallelization</b> . . . . .	37
IBM Directives . . . . .	37
OpenMP Directives . . . . .	37
Countable Loops . . . . .	38
Reduction Operations in Parallelized Loops . . . . .	39
Shared and Private Variables in a Parallel Environment . . . . .	40
Using Pragmas to Control Parallel Processing . . . . .	41

<b>Chapter 6. The C Language</b>	45
Lexical Elements of C	45
Tokens	45
Comments	46
Identifiers	47
Constants	48
Identifier Behavior in Your Program	53
Scope of Identifier Visibility	53
Program Linkage Between Identifiers	54
Storage Duration	57
Name Spaces	57
Preprocessor Directives	58
Preprocessing Operations	59
Preprocessor Macros	59
Conditional Compilation Directives	60
Declarations Overview	61
Block Scope Data Declarations	62
Initialization	62
Storage	62
File Scope Data Declarations	63
Declarators	64
Storage Class Specifiers	65
Initializers	65
Type Specifiers	66
Expressions and Operators	67
Operator Precedence and Associativity	67
Operands	69
lvalues	70
Types of Expressions	70
Constant Expressions	71
Function Calls	72
Implicit Type Conversions	74
Integral Promotions	74
Standard Type Conversions	74
Arithmetic Conversions	76
Functions	77
Calling Functions and Passing Arguments	77
C Language Levels	78
Basic Data Types	79
char	79
float, double	80
int, long, short	81
enum	82
void	85
Derived Data Types	86
Arrays	86
Pointers	90
struct (Structures)	95
union (Unions)	103
Incomplete Types	106
auto	106
extern	109
register	111
static	112
typedef	115
Data Type Qualifiers	115



Expression Operators . . . . .	117
Operator Precedence and Associativity Table . . . . .	117
Primary Operators . . . . .	118
Unary Operators . . . . .	120
Binary Operators . . . . .	124
Conditional Operator (?) . . . . .	131
Assignment Operators . . . . .	133
Comma Operator ( , ) . . . . .	136
Arithmetic Conversions Table . . . . .	137
Functions . . . . .	138
Function Declarations . . . . .	138
Function Definitions . . . . .	139
main() Function . . . . .	144
Program Statement Keywords . . . . .	145
break . . . . .	145
continue . . . . .	147
do . . . . .	148
Expressions . . . . .	149
for . . . . .	149
goto . . . . .	151
if / else . . . . .	152
Null Statement . . . . .	153
return . . . . .	154
switch . . . . .	155
while . . . . .	158
Statement Labels . . . . .	158
Statement Blocks . . . . .	159
Example of Initialization within Statement Blocks . . . . .	159
C Programming Character Set . . . . .	160
Escape Sequences for Non-Printable Characters . . . . .	161
Reserved Keywords . . . . .	161
Differences Between C Language Levels . . . . .	162
Conflicts Between extended C and Other Levels . . . . .	162
Extensions to RT C Provided by extended C . . . . .	164
Exceptions to ansi C Addressed by classic C . . . . .	164
saal2 C Deviations from SAA Level 2 C . . . . .	167
Arithmetic Conversions for extended Level C . . . . .	167
Summary of C Language Level Conflicts . . . . .	170
<b>Chapter 7. Writing C Programs . . . . .</b>	<b>173</b>
Creating and Naming a C Source File . . . . .	174
File-Naming Conventions . . . . .	174
Internal Structure of a C Program . . . . .	174
Example of a Simple C Program . . . . .	175
Example of a C Program Comprised of Two Source Files . . . . .	176
External Structure of a C Program . . . . .	177
Specifying Path Names for Include Files . . . . .	178
Using a Full Path Name to Imbed Files . . . . .	178
Using a Relative Path Name to Imbed Files . . . . .	178
Directory Search Sequence for Include Files Using Relative Path Names . . . . .	178
Using Memory Heaps in a Program . . . . .	179
Memory Management Functions . . . . .	179
Managing Memory with Multiple Heaps . . . . .	182
Types of Memory . . . . .	183
Debugging Memory Heaps . . . . .	183
Changing the Default Heap Used in a Program . . . . .	185

Creating and Using a Fixed Size Heap . . . . .	186
Creating and Using an Expandable Heap . . . . .	188
Expanding Your Heap . . . . .	189
Shrinking Your Heap . . . . .	189
Example of Creating and Using a User Heap . . . . .	190
Example of Creating and Using a Shared-Memory User Heap . . . . .	191
Debugging Programs with Heap Memory . . . . .	195
Writing Optimized Program Source Code . . . . .	197
Variables . . . . .	197
Pointers . . . . .	198
Functions . . . . .	199
Function Arguments . . . . .	199
Expressions . . . . .	199
Critical Loops . . . . .	200
Conversions . . . . .	201
Arithmetic Constructions . . . . .	201
Using Inlined Components. . . . .	202
<b>Chapter 8. Using C for AIX with Other Programming Languages . . . . .</b>	<b>207</b>
Interlanguage Calling Conventions. . . . .	207
Corresponding Data Types . . . . .	207
Special Treatment of Character and Aggregate Data . . . . .	208
Using the Subroutine Linkage Conventions in Interlanguage Calls . . . . .	209
Interlanguage Calls - Parameter Passing . . . . .	210
Interlanguage Calls - Call by Reference Parameters . . . . .	210
Interlanguage Calls - Call by Value Parameters . . . . .	211
Interlanguage Calls - Rules for Passing Parameters by Value . . . . .	211
Interlanguage Calls - Pointers to Functions . . . . .	212
Interlanguage Calls - Function Return Values. . . . .	213
Interlanguage Calls - Stack Floor . . . . .	213
Interlanguage Calls - Stack Overflow . . . . .	213
Interlanguage Calls - Traceback Table . . . . .	214
Interlanguage Calls - Type Encoding and Checking . . . . .	214
Sample Program: C Calling Fortran . . . . .	214
<b>Appendix A. Compiler Options . . . . .</b>	<b>217</b>
Resolving Conflicting Compiler Options . . . . .	217
Compiler Options and Their Defaults . . . . .	218
Lists of Compiler Options by Functional Groupings. . . . .	226
Options that Specify Compiler Characteristics . . . . .	226
Options that Specify Debugging Features . . . . .	227
Options that Specify Preprocessor Options . . . . .	228
Options that Specify Compiler Output . . . . .	228
Options that Specify the Compiler Object Code Produced . . . . .	229
Options that Specify Linkage Options. . . . .	230
Compiler Options Reference . . . . .	231
# . . . . .	231
32, 64 . . . . .	231
aggcopy . . . . .	232
alias . . . . .	233
align . . . . .	234
ansialias . . . . .	236
arch . . . . .	237
assert . . . . .	238
attr . . . . .	238
B . . . . .	239

bitfields . . . . .	240
brtl . . . . .	240
bstatic, bdynamic . . . . .	241
C . . . . .	242
c . . . . .	242
cache . . . . .	243
chars . . . . .	244
check . . . . .	245
compact . . . . .	246
cpluscmt . . . . .	247
D . . . . .	250
datalocal, dataimported . . . . .	251
dbxextra . . . . .	252
digraph . . . . .	252
dollar . . . . .	253
dpcl . . . . .	253
E . . . . .	253
enum . . . . .	255
extchk . . . . .	258
f . . . . .	259
F . . . . .	259
fdpr . . . . .	260
flag . . . . .	261
float . . . . .	261
flttrap . . . . .	264
fold . . . . .	265
fullpath . . . . .	266
G . . . . .	266
g . . . . .	267
genpcomp. . . . .	267
genproto . . . . .	268
halt . . . . .	269
heapdebug . . . . .	270
hsflt . . . . .	271
hssngl . . . . .	272
I . . . . .	272
idirfirst . . . . .	273
ignerrno . . . . .	274
ignprag . . . . .	274
info . . . . .	275
initauto . . . . .	276
inlglue . . . . .	277
inline . . . . .	277
ipa . . . . .	279
isolated_call . . . . .	284
L . . . . .	285
l . . . . .	286
langlvl . . . . .	286
ldb128, longdouble . . . . .	289
libansi . . . . .	290
linedebug . . . . .	291
list . . . . .	291
listopt . . . . .	292
longlit . . . . .	292
longlong . . . . .	293
M . . . . .	294

ma	295
macpstr	295
maf	297
makedep	298
maxerr	299
maxmem	300
mbsc, dbsc	301
noprint	301
O, optimize	302
o	305
once	306
P	307
p	308
pascal	308
pdf1, pdf2	309
pg.	311
phsinfo	312
proclcal, procimported, procunknown	312
proto.	313
Q	314
r	316
rndsngl	316
ro	317
roconst	317
rrm	318
S	319
showinc	320
smp	320
source	322
spill	323
spnans	323
srcmsg	324
statsym.	324
stdinc	325
strict	326
strict_induction	327
syntaxonly	327
suppress	328
t	329
tabsize	329
tbtable	330
threaded	331
tune	331
U	332
unroll	333
upconv	334
usepcomp.	335
v	336
W	336
w	337
warn64	338
xcall	338
xref	339
y	339

**Appendix B. 32-bit to 64-bit Migration Considerations** . . . . . 341

<b>Appendix C. Operating System Migration Considerations</b> . . . . .	345
<b>Appendix D. Preprocessor Directives and Related Information</b> . . . . .	347
List of Standard Preprocessor Directives . . . . .	347
# (Null) Preprocessor Directive . . . . .	347
#define Preprocessor Directive . . . . .	348
#if, #elif Preprocessor Directives . . . . .	352
#else Preprocessor Directive . . . . .	353
#endif Preprocessor Directive . . . . .	353
#error Preprocessor Directive . . . . .	354
#ifdef Preprocessor Directive . . . . .	354
#ifndef Preprocessor Directive . . . . .	355
#include Preprocessor Directive . . . . .	356
#line Preprocessor Directive . . . . .	357
#undef Preprocessor Directive . . . . .	358
Predefined Preprocessor Macros . . . . .	359
Examples of Predefined Macros in a Program . . . . .	362
#pragma Preprocessor Directives . . . . .	363
#pragma alloca Preprocessor Directive . . . . .	365
#pragma chars Preprocessor Directive . . . . .	365
#pragma comment Preprocessor Directive . . . . .	366
#pragma disjoint Preprocessor Directive . . . . .	366
#pragma execution_frequency Preprocessor Directive . . . . .	367
#pragma hdrfile Preprocessor Directive . . . . .	368
#pragma hdrstop Preprocessor Directive . . . . .	369
#pragma info Preprocessor Directive . . . . .	370
#pragma isolated_call Preprocessor Directive . . . . .	371
#pragma langlvl Preprocessor Directive . . . . .	373
#pragma leaves Preprocessor Directive . . . . .	373
#pragma map Preprocessor Directive . . . . .	374
#pragma option_override Preprocessor Directive . . . . .	374
#pragma options Preprocessor Directive . . . . .	375
#pragma reachable Preprocessor Directive . . . . .	376
#pragma strings Preprocessor Directive . . . . .	376
Preprocessor Macro Operators . . . . .	377
# Preprocessor Macro Operator . . . . .	377
## Preprocessor Macro Operator . . . . .	378
/**/ Preprocessor Macro Operator . . . . .	379
<b>Appendix E. Parallel Processing Facilities</b> . . . . .	381
#pragma Preprocessor Directives for Parallel Processing . . . . .	381
#pragma ibm critical Preprocessor Directive . . . . .	382
#pragma ibm independent_calls Preprocessor Directive . . . . .	383
#pragma ibm independent_loop Preprocessor Directive . . . . .	384
#pragma ibm iterations Preprocessor Directive . . . . .	384
#pragma ibm parallel_loop Preprocessor Directive . . . . .	385
#pragma ibm permutation Preprocessor Directive . . . . .	385
#pragma ibm schedule Preprocessor Directive . . . . .	386
#pragma ibm sequential_loop Preprocessor Directive . . . . .	387
#pragma omp parallel Preprocessor Directive . . . . .	388
#pragma omp for Preprocessor Directive . . . . .	389
#pragma omp parallel for Preprocessor Directive . . . . .	393
#pragma omp sections Preprocessor Directive . . . . .	393
#pragma omp parallel sections Preprocessor Directive . . . . .	394
#pragma omp single Preprocessor Directive . . . . .	395
#pragma omp master Preprocessor Directive . . . . .	395

#pragma omp critical Preprocessor Directive . . . . .	396
#pragma omp barrier Preprocessor Directive . . . . .	397
#pragma omp atomic Preprocessor Directive . . . . .	397
#pragma omp flush Preprocessor Directive . . . . .	398
#pragma omp ordered Preprocessor Directive . . . . .	399
#pragma omp threadprivate Preprocessor Directive . . . . .	399
Built-in Functions Used for Parallel Processing . . . . .	400
Run-time Options for Parallel Processing . . . . .	402
OpenMP Run-time Options for Parallel Processing . . . . .	404
<b>Appendix F. C for AIX Debug Functions . . . . .</b>	<b>407</b>
_debug_calloc - Allocate and Initialize Memory . . . . .	407
_debug_free - Free Allocated Memory . . . . .	408
_debug_heapmin - Free Unused Memory in the Default Heap . . . . .	410
_debug_malloc - Allocate Memory . . . . .	412
_debug_memcpy - Copy Bytes . . . . .	413
_debug_memmove - Copy Bytes . . . . .	415
_debug_memset - Set Bytes to Value . . . . .	416
_debug_realloc - Reallocate Memory Block . . . . .	417
_debug_strcat - Concatenate Strings . . . . .	419
_debug_strcpy - Copy Strings . . . . .	421
_debug_strncat - Concatenate Strings . . . . .	422
_debug_strncpy - Copy Strings . . . . .	423
_debug_strnset - Set Characters in String . . . . .	425
_debug_strset - Set Characters in String . . . . .	426
_debug_ucalloc - Reserve and Initialize Memory from User Heap . . . . .	428
_debug_uheapmin - Free Unused Memory in User Heap . . . . .	430
_debug_umalloc - Reserve Memory Blocks from User Heap . . . . .	431
<b>Appendix G. Built-in Functions for PowerPC Processors . . . . .</b>	<b>435</b>
<b>Appendix H. RISC System/6000 Alignment Rules . . . . .</b>	<b>437</b>
Alignment Rules for Nested Aggregates . . . . .	438
Packed Alignment Rules . . . . .	438
MacIntosh and Twobyte Alignment Rules . . . . .	440
__align Specifier . . . . .	442
<b>Appendix I. Implementation Dependencies Overview . . . . .</b>	<b>445</b>
C for AIX Compiler Limits . . . . .	445
Implementation-Defined Behavior . . . . .	445
Implementation Dependency - Translation (F.3.1) . . . . .	446
Implementation Dependency - Environment (F.3.2) . . . . .	447
Implementation Dependency - Identifiers (F.3.3) . . . . .	448
Implementation Dependency - Characters (F.3.4) . . . . .	448
Implementation Dependency - Integers (F.3.5) . . . . .	449
Implementation Dependency - Floating Point Types (F.3.6) . . . . .	451
Implementation Dependency - Arrays and Pointers (F.3.7) . . . . .	452
Implementation Dependency - Registers (F.3.8) . . . . .	452
Implementation Dependency - Structures, Unions, Enumerations, Bit Fields (A.6.3.9) (F.3.9) . . . . .	453
Implementation Dependency - Qualifiers (F.3.10) . . . . .	453
Implementation Dependency - Declarators (F.3.11) . . . . .	454
Implementation Dependency - Statements (F.3.12) . . . . .	454
Implementation Dependency - Preprocessing Directives (F.3.13) . . . . .	454
Implementation Dependency - Library Functions (F.3.14) . . . . .	455
Implementation Dependency - Locale-Specific Behavior (F.4) . . . . .	459
Type Conversions . . . . .	461

Synchronization of Stores and Loads to I/O Space . . . . .	464
Example of Multiple Writes to a Single Register . . . . .	465
Example of Reading and Writing to Mapped-to-I/O Space . . . . .	465
<b>Appendix J. C for AIX and XL C Compatibility . . . . .</b>	<b>467</b>
<b>Appendix K. National Languages Support in the C for AIX Compiler. . . . .</b>	<b>469</b>
Converting Files Containing Multibyte Data to New Code Pages . . . . .	469
Where Multibyte Characters Are Supported . . . . .	469
<b>Appendix L. C for AIX Files. . . . .</b>	<b>473</b>
etc/vac.cfg - Default Configuration File . . . . .	474
vac.cfg.41 Compiler Configuration File . . . . .	474
vac.cfg.43 Compiler Configuration File . . . . .	477
<b>Appendix M. ASCII Character Set . . . . .</b>	<b>483</b>
<b>Appendix N. Problem Solving . . . . .</b>	<b>487</b>
Message Catalog Errors . . . . .	487
Correcting Page Space Errors During Compilation . . . . .	487
<b>Appendix O. Glossary . . . . .</b>	<b>489</b>
A . . . . .	489
B . . . . .	489
C . . . . .	490
D . . . . .	491
E . . . . .	492
F . . . . .	492
G . . . . .	493
H . . . . .	493
I . . . . .	493
K . . . . .	494
L . . . . .	494
M . . . . .	495
N . . . . .	495
O . . . . .	495
P . . . . .	496
R . . . . .	496
S . . . . .	496
T . . . . .	497
U . . . . .	497
V . . . . .	498
W . . . . .	498
Z . . . . .	498





---

## Notices

Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter 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:

Director of Licensing,  
Intellectual Property & Licensing,  
International Business Machines Corporation,  
North Castle Drive, MD - NC119,  
Armonk, New York 10504-1785,  
U.S.A.

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 IBM Canada Ltd., Department 071, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7, Canada. Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

This publication may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may 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.

IBM may change this publication, the product described herein, or both.

---

## Trademarks and Service Marks

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

AIX  
AIXwindows  
C Set ++  
IBM  
OS/2  
POWER  
POWER2  
PowerPC  
RS/6000

Windows is a trademark or registered trademark of Microsoft Corporation in the U.S. and/or other countries.

UNIX is a registered trademark in the U.S. and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names, which may be denoted by a double asterisk(\*\*), may be trademarks or service marks of others.



---

## About this Information

This information describes the IBM C for AIX licensed program product, intended for use with the AIX Version 4 Operating System environment.

You will find information on using the C for AIX compiler product to compile, link, and run programs coded in the C language. Sections describe how to both write and use compiler options to better optimize programs compile with the C for AIX product. Also included is C language reference information.

### Highlighting Conventions

This information uses the following text-highlighting conventions:

#### **Bold Font**

- Names of operating system commands
- Names of compiler options
- Names of language keywords
- Directory paths

#### Monospaced Font

- Information that you should actually type
- Examples of code
- Examples of text or system messages that you might see displayed on the screen

#### *Italic Font*

- Variables for which you will substitute actual names

---

## Related Reading

All C for AIX information is available online and can be viewed with an HTML browser. You may also want refer to the following publications for additional information:

### IBM Publications

- *AIX Version 4 System User's Guide: Operating System and Devices* (SC23-2544)  
Describes the AIX Version 4 Operating System for novice system users. It describes how to run commands, handle processes, files and directories, printing, and working with the AIXwindows Desktop. It also introduces system commands for securing files, using storage media, and customizing environment files.
- *AIX Version 4 Getting Started* (SC23-2527)  
Contains information for users who have little or no experience with the AIX operating system. It introduces basic system commands covering tasks such as starting and stopping the system, using a keyboard or mouse, logging in and out, identifying and using the various user interfaces, and running basic file commands.
- *AIX Version 4 Commands Reference* (SBOF-1851)  
A collection of volumes that contain descriptions and examples of AIX commands and their available flags.
- *AIX Version 4 General Programming Concepts* (SC23-2533 and SC23-2490)  
Discusses the operating system from a programming perspective.
- *AIX Version 4 Technical Reference, Volumes 1 and 2: Base Operating System and Extensions* (SC23-2614 and SC23-2615)  
Provides reference information about system calls, subroutines, macros, and statements associated with the AIX base operating system runtime services and device services.

## Non-IBM Publications

The C language is a well-established programming language. The following standards describe it:

- *ANSI/ISO-IEC 9899-1990[1992]*  
Presents the ANSI/ISO standard for the C language. This document has officially replaced American National Standard for Information Systems-Programming Language C (X3.159-1989) as the ANSI C standard, and is technically equivalent to ANSI X3.159-1989.
- *ISO/IEC 9899:1990(E)*  
Presents the International Standards Organization (ISO) standard for the C language.
- *Federal Information Processing Standards Publication C (FIPS PUB 160)*  
Presents the Federal Information Processing Standard (FIPS) for the C language.

---

# Chapter 1. Introducing C for AIX

The C for AIX product is an IBM licensed program that operates in the AIX Version 4 Operating System environment. Features of the C for AIX product include:

- Ability to compile in either 64- or 32-bit modes. Programs compiled in 64-bit mode can only be run on 64-bit CPUs using AIX 4.3 or higher. Programs compiled in 32-bit mode can be run on either 64- or 32-bit CPUs using AIX 4.2 or higher.
- Programming support for parallel processing architectures:
  - SMP automatic and explicit parallelization support
  - OpenMP Application Program Interface support
- Conformance to the following industry standards for compiling C language source code:
  - The Federal Information Processing Standard (FIPS) PUB 160 C language
  - The American National Standard for Information Systems (ANSI) and International Standards Organization (ISO) standard ANSI/ISO-IEC 9899-1990[1992] for the C programming language
  - The International Standards Organization (ISO) standard ISO/IEC 9899:1990(E) for the C programming language
  - Conformance to IBM Systems Application Architecture (SAA) Common Programming Interface C language definition, described by the document *Systems Application Architecture Common Programming Interface C Reference - Level 2*. SAA Level 2 is an IBM definition of the C language that allows programmers to develop applications that can be easily transported across different SAA environments. It specifies several features of the C language that the ANSI C standard designates as implementation-defined.
- Compiler options to provide support for different levels and features of the C language
- Compiler options to enable various levels of optimization for generated object code
- Unicode character support lets you use characters not in the basic character set to describe identifiers, character constants, and string literals.
- Memory debug routines
- **xldb** and IBM Distributed Debugger (**idebug**) graphical debugger tools
- HTML-based product help and reference information



---

## Chapter 2. Setting Up the C for AIX Compilation Environment

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

---

### Setting Environment Variables to Select 64- or 32-bit Compilation Modes

The OBJECT\_MODE environment variable, if it exists, can set the default compilation mode. Permissible values for the OBJECT\_MODE environment variable are:

32	Sets the compiler to generate and/or use 32-bit objects.
64	Sets the compiler to generate and/or use 64-bit objects.
32_64	Sets the compiler 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.

See “Specifying Compiler Options for Architecture-Specific, 32- or 64-bit Compilation” on page 14 for more information.

---

### Setting 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.

See “Run-time Options for Parallel Processing” on page 402 and “OpenMP Run-time Options for Parallel Processing” on page 404 for more information.

---

### Setting 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:

LANG	Specifies the national language for message and help files.
NLSPATH	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.

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 (**cs**h). 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 *AIX Version 4 System User's Guide: Operating System and Devices*. The AIX international language facilities are described in the *AIX General Programming Concepts for IBM RISC System/6000*.

---

## Setting Environment Variables in bsh, ksh, or sh Shells

To set the environment variables from the Bourne shell or Korn shell, use the following commands:

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

---

## Setting Environment Variables in csh Shell

To set the environment variables from the C shell, use the following commands:

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

"Specifying Compiler Options for Architecture-Specific, 32- or 64-bit Compilation" on page 14

"Run-time Options for Parallel Processing" on page 402

"OpenMP Run-time Options for Parallel Processing" on page 404



---

## Chapter 3. Using the C for AIX Compiler

You can use the C for AIX product as a C compiler for files with a `.c` (small c) suffix. The compiler processes your text-based C program source files to create an executable object module.

The `cc` command is a traditional UNIX operating system command for invoking the C compiler. Other compiler modes and their invocation commands described in this and related pages are specific to the C for AIX compiler.

<b>Note: Use of the <code>xlc</code> Command in this Information</b>
--

Throughout these information panels, the <code>xlc</code> command is used to describe the actions of the compiler.
--

In most cases, you should use the <code>xlc</code> command to compile your C source files.
--

The <code>xlc_r</code> and <code>xlc128</code> commands specify additional libraries, macros, or options that are not automatically included or set by the <code>xlc</code> command. Besides these differences, these commands may be considered functionally equivalent, so that any mention of one in this book implies the other. This is also true for the <code>cc</code> , <code>cc_r</code> and <code>cc128</code> commands.
---

### Related Reading

“Compiler Modes”

“Compiler Options” on page 10

“Types of Input Files” on page 7

“Types of Output Files” on page 8

“Compiler Message and Listing Information” on page 18

“Invoking the Linkage Editor” on page 9

“Invoking the Compiler” on page 8

“Specifying Compiler Options on the Command Line” on page 10

“Specifying Compiler Options in Your Program Source Files” on page 12

“Specifying Compiler Options in a Configuration File” on page 13

“Compiler Options and Their Defaults” on page 218

---

## Compiler Modes

There are several forms of the C for AIX compiler command to support various version levels of the C language. Normally, you should use the `xlc` command for compiling your source files. You can, however, use other forms of the command if your particular environment and file systems require it.

The basic compiler invocations are:

<code>x1c</code>	Invokes the compiler for C source files with a default language level of <b>ansi</b> , and specifies compiler option <b>-qansialias</b> to allow type-based aliasing. Use this invocation for new C programs.
<code>cc</code>	Invokes the compiler for C source files with a default language level of <b>extended</b> and compiler options <b>-qnor</b> and <b>-qnorconst</b> (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.
<code>c89</code>	Invokes the compiler for C source files, with a default language level of <b>ansi</b> , and specifies compiler options <b>-qansialias</b> (to allow type based aliasing) and <b>-qolonglong</b> (disabling use of <b>long long</b> ), and sets <b>-D_ANSI_C_SOURCE</b> (for ANSI-conformant headers). Use this invocation for strict conformance to the ANSI standard.

C for AIX provides variations on the four basic compiler invocations. These variations are described below:

**xlc128**  
**cc128** All **128**-suffixed invocation commands are functionally similar to their corresponding base compiler invocations. They specify the **-qldbl128** option, which increases the length of **long double** types in your program from 64 to 128 bits.

**xlc\_r**  
**cc\_r** All **\_r**-suffixed invocations are functionally similar to their corresponding base compiler invocations, but set the macro name **-D\_THREAD\_SAFE** and invoke the added compiler options:

- **-L/usr/lib/threads**
- **-Lusr/lib/dce**
- **-lc\_r**
- **-lpthreads**
- **-qthreaded**

Use the **\_r**-suffixed invocations when compiling with the **-qsmp** compiler option or if you want to create either Posix or AIX DCE threaded applications.

**xlc\_r4**  
**cc\_r4** Use **\_r4**-suffixed invocations to 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.

On AIX 4.3, use **\_r7**-suffixed invocations to compile and link applications conforming to DRAFT 7 of the Posix threads standard. Otherwise, the compiler will by default compile and link applications conforming to the current Posix threads standards.

**xlc\_r7**  
**cc\_r7** On AIX 4.3, use **\_r7**-suffixed invocations to compile and link applications conforming to DRAFT 7 of the Posix threads standard. Otherwise, the compiler will by default compile and link applications conforming to the current Posix threads standards.

#### **Related Reading**

"C Language Levels" on page 78

"Invoking the Compiler" on page 8

"ansialias" on page 236

"D" on page 250

"L" on page 285

"I" on page 286

"longlong" on page 293

"ro" on page 317

"roconst" on page 317

---

## Types of Input Files

You can input the following types of files to the C for AIX compilers.

### C Source Files

These are files containing a C source module. The source file must have a **.c** (lowercase c) suffix, for example, `mysource.c`.

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 **xlc**. 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 **xlc** command preprocesses and compiles all the specified source files. Although you will usually want to use this default, you can use the **xlc** 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 **xlc** 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 an **.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 an **.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 Version 4 Commands Reference*, and the description of **a.out** file format in the *AIX Version 4 Files Reference* for more information.

### Related Reading

“Types of Output Files” on page 8

“E” on page 253

“P” on page 307

---

## Types of Output Files

You can specify the following types of output files when invoking the C for AIX compiler.

- Executable File** By default, executable files are named a.out. To name the executable file something else, use the **-ofile\_name** option with the invocation command. This option creates an executable file with the name you specify as *file\_name*. The name you specify can be a relative or absolute path name for the executable file.
- Object Files** The format of the a.out file is described in the *AIX Version 4 Files Reference*. Object files must have an **.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. .
- Assembler Files** You can link-edit the object files later into a single executable file using the **xlc** command. Assembler files must have an **.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 an **.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.
- Listing Files** A preprocessed source file, *file\_name.i*, is produced for each source file, *file\_name.c*. Listing files have an **.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 an **.lst** extension) as the source file from which it was produced.
- Target File** Output files associated with the **-M** option have an **.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 file with a **.c** or **.i** suffix. **.u** files are not created for any other files (unless you use the **-+** option so other file suffixes are treated as **.c** files).

### Related Reading

“Types of Input Files” on page 7 “**c**” on page 242 “**M**” on page 294

“**o**” on page 305 “**P**” on page 307

“**S**” on page 319

“**noprint**” on page 301

---

## Invoking the Compiler

All forms of the C for AIX compiler are invoked using the following syntax, where *invocation* can be replaced with any valid C for AIX compiler mode invocation command:

```
>>—invocation—↓—————><
                |—————|—————|
                |command_line_options|input_files|
```

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

### Related Reading

“Chapter 7. Writing C Programs” on page 173  
“Compiler Modes” on page 5  
“Specifying Compiler Options on the Command Line” on page 10  
“Compiler Options and Their Defaults” on page 218  
“Message Severity Levels and Compiler Response” on page 20  
“Compiler Return Codes” on page 20  
“Compiler Message Format” on page 21  
“c” on page 242

---

## Invoking the Linkage Editor

The linkage editor link-edits all of the 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**, or **-#**.

### Input Files

Object files, library files, and unstripped executable files serve as input to the linkage editor.

### Object Files

Object files must have a **.o** suffix, for example, **year.o**.

### Library Files

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 **xc** command, where *file\_name* is the name you want to give to the executable file. If you use the **-ofile\_name** option, the resulting executable file is called *file\_name*.

### Using the ld Command

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 Reading

- “Chapter 7. Writing C Programs” on page 173
- “Invoking the Compiler” on page 8 “#” on page 231
- “`c`” on page 242
- “`E`” on page 253
- “`o`” on page 305
- “`P`” on page 307

---

## 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 your source program
- in a configuration file

When specifying compiler options in more than one of the above locations, it is possible for option conflicts and incompatibilities to occur. C for AIX resolves these conflicts and incompatibilities in a consistent fashion, as described in “Resolving Conflicting Compiler Options” on page 217.

#### Related Reading

- “Invoking the Compiler” on page 8
- “Specifying Compiler Options on the Command Line”
- “Specifying Compiler Options in Your Program Source Files” on page 12
- “Specifying Compiler Options in a Configuration File” on page 13
- “Resolving Conflicting Compiler Options” on page 217
- “Compiler Options and Their Defaults” on page 218

## Specifying Compiler Options on the Command Line

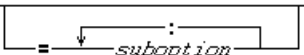
Most options specified on the command line override both the default settings of the option and options set in the configuration file. Similarly, most options specified on the command line are in turn overridden by options set in the source file. Options that do not follow this scheme are listed in “Resolving Conflicting Compiler Options” on page 217.

There are two kinds of command-line options:

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

### -q Options

```
>> -qoption_keyword [ = suboption ] <<
```



Command-line options in the **-qoption\_keyword** format are similar to on and off switches. If the option is specified more than once, the last instance is recognized by the compiler. For example, **-qsource** turns on the source option to produce a compiler listing; **-qnosource** turns off the source option, so no source listing is produced. For example:

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

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

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

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

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

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

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

## Flag Options

The compilers available on AIX systems use a number of common conventional flag options. The C for AIX compiler 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.

The C for AIX compiler 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:

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

where new.cfg is a custom configuration file.

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

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

has the same effect as:

```
xlc -O -c -v file.c
```

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

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

```
xlc -Ovotest test.c
```

has the same effect as:

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

Most flag options are a single letter, but some are two letters. Note that **-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 Reading

“Compiler Options” on page 10

“Invoking the Compiler” on page 8

“Specifying Compiler Options in Your Program Source Files”

“Specifying Compiler Options in a Configuration File” on page 13

“Resolving Conflicting Compiler Options” on page 217

“Compiler Options and Their Defaults” on page 218

## Specifying Compiler Options in Your Program Source Files

To specify compiler options in your program source files, use the preprocessor directive:

```
#pragma options compiler_options
```

If you specify more than one compiler option, 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
```

Options specified 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 other **#pragma** preprocessor directives, see “**#pragma** Preprocessor Directives” on page 363 and “**#pragma** Preprocessor Directives for Parallel Processing” on page 381.



## Related Reading

- “Compiler Options” on page 10
- “Invoking the Compiler” on page 8
- “Specifying Compiler Options on the Command Line” on page 10
- “Specifying Compiler Options in a Configuration File”
- “Resolving Conflicting Compiler Options” on page 217
- “Compiler Options and Their Defaults” on page 218
- “#pragma Preprocessor Directives” on page 363
- “#pragma Preprocessor Directives for Parallel Processing” on page 381

## Specifying 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 programs. You can make entries to this file to support specific compilation requirements or to support other 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” on page 217.

**Tailoring a Configuration File** The default configuration file is `/etc/vac.cfg`.

You can copy this file and make changes to the copy to support specific compilation requirements or to support other 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 **xlc** compiler invocation command, add **-qnor0** to the **xlc** stanza in your copied version of the configuration file.

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

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

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

## Configuration File Attributes

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

<b>as</b>	Path name to be used for the assembler. The default is <b>/bin/as</b> .
<b>asopt</b>	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 <b>as</b> stanza. The string is formatted for the AIX <b>getopt()</b> subroutine as a concatenation of flag letters, with a letter followed by a colon (:) if the corresponding flag takes a parameter.
<b>cppcode</b>	Path name to be used for the code generation phase of the compiler. The default is <b>/usr/vac/exe/xlCcode</b> .
<b>ccomp</b>	C Front end. The default is <b>/usr/vac/exe/xlcentry</b> .
<b>codeopt</b>	List of options for the code-generation phase of the compiler.
<b>cppopt</b>	List of options for the lexical analysis phase of the compiler.
<b>crt</b>	Path name of the object file passed as the first parameter to the linkage editor. If you do not specify either the <b>-p</b> or the <b>-pg</b> option, the <b>crt</b> value is used. The default is <b>/lib/crt0.o</b> .
<b>csuffix</b>	Suffix for source programs. The default is <b>c</b> (lowercase c).
<b>dis</b>	Path name of the disassembler. The default is <b>/usr/vac/exe/dis</b> .

<b>gcrt</b>	Path name of the object file passed as the first parameter to the linkage editor. If you specify the <b>-pg</b> option, the gcrt value is used. The default is <b>/lib/grt0.o</b> .
<b>inline</b>	Path name to be used for the inlining phase of the compiler. The default is <b>/usr/vac/exe/xlCinline</b> .
<b>inlineopt</b>	List of options for the inlining phase of the compiler.
<b>ld</b>	Path name to be used to link C programs. The default is <b>/bin/ld</b> .
<b>ldopt</b>	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 <b>getopt()</b> subroutine as a concatenation of flag letters, with a letter followed by a colon (:).
<b>libraries2</b>	Library options, separated by commas, that the compiler passes as the last parameters to the linkage editor. <b>libraries2</b> specifies the libraries that the linkage editor is to use at link-edit time for both profiling and nonprofiling. The default is empty.
<b>mcrt</b>	Path name of the object file passed as the first parameter to the linkage editor if you have specified the <b>-p</b> option. The default is <b>/lib/mcrt0.o</b> .
<b>options</b>	A string of option flags, separated by commas, to be processed by the compiler as if they had been entered on the command line.
<b>osuffix</b>	The suffix for object files. The default is <b>.o</b> .
<b>proflibs</b>	Library options, separated by commas, that the compiler passes to the linkage editor when profiling options are specified. <b>proflibs</b> specifies the profiling libraries used by the linkage editor at link-edit time. The default is <b>-L/lib/profiled</b> and <b>-L/usr/lib/profiled</b> .
<b>ssuffix</b>	The suffix for assembler files. The default is <b>.s</b> .
<b>use</b>	Values for attributes are taken from the named stanza and from the local stanza. For single-valued attributes, values in the <b>use</b> 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.
<b>xc</b>	The path name of the <b>xc</b> compiler component. The default is <b>/usr/vac/bin/xc</b> .

### Related Reading

- “Compiler Options” on page 10
- “Invoking the Compiler” on page 8
- “Specifying Compiler Options on the Command Line” on page 10
- “Specifying Compiler Options in Your Program Source Files” on page 12
- “Resolving Conflicting Compiler Options” on page 217
- “Compiler Options and Their Defaults” on page 218
- “etc/vac.cfg - Default Configuration File” on page 474
- “F” on page 259
- “L” on page 285
- “p” on page 308
- “pg” on page 311

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

You can use C for AIX 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:

<b>OBJECT_MODE Setting</b>	<b>User-selected compilation-mode behavior, unless overridden by configuration file or command-line options</b>
not set	32-bit compiler mode.
32	32-bit compiler mode.

**OBJECT\_MODE** User-selected compilation-mode behavior, unless overridden by configuration file or command-line options

Setting	
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** option to **com**, and sets **-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** option 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 Reading

- “Compiler Options” on page 10
- “Invoking the Compiler” on page 8
- “Chapter 2. Setting Up the C for AIX Compilation Environment” on page 3
- “Specifying Compiler Options in Your Program Source Files” on page 12
- “Specifying Compiler Options in a Configuration File” on page 13
- “Acceptable Compiler Mode and Processor Architecture Combinations”
- “Resolving Conflicting Compiler Options” on page 217
- “Compiler Options and Their Defaults” on page 218
- “32, 64” on page 231
- “arch” on page 237
- “tune” on page 331

## 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 compiler mode.
- **-q64** selects 64-bit compiler mode.
- **-qarch** selects the general family processor architecture for which code (instructions) should be generated. Certain **-qarch** settings will produce code that will run *only* on RS/6000 systems that support *all* of the instructions generated by the compiler in response to the chosen **-qarch** settings.
- **-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 families of RS/6000 machines:

- POWER
- POWER2
- PowerPC

Each of these families have a different instruction set but share a common subset of instructions. The POWER2 instruction set is a superset of the POWER instructions set. The PowerPC instruction set includes additional instructions not available on POWER systems but does not support all of the POWER instruction set. It also includes some but not all of the POWER2 instructions not available in the POWER instruction set. Further, some features found in the POWER2 instruction set may or may not be implemented on particular PowerPC processors. These optional feature groups are:

- support for the graphics instruction group
- support for the sqrt instruction group
- support for 64-bit support (**-q64** compiler option)

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=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**.

Processor	graphics support	sqrt support	64-bit support
603	yes	no	no
604	yes	no	no
rs64a	no	no	yes
rs64b	yes	yes	yes
pwr3	yes	yes	yes

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 table below. If you specify incompatible combinations of these options, the compiler will assume its own option selections, as described in “Specifying Compiler Options for Architecture-Specific, 32- or 64-bit Compilation” on page 14.

-qarch option	Predefined Macro	Available -qtune options	Default -qtune suboption	-q32	-q64
<b>com</b>	<code>_ARCH_COM</code>	DEFAULT (not selectable)  pwr pwr2 pwr2s pwr3 pwrx p2sc 601 602 603 604 403 rs64a rs64b	pwr2	yes	yes
<b>pwr</b>	<code>_ARCH_PWR</code>	pwr pwr2 pwr2s pwrx p2sc 601	pwr2	yes	no
<b>pwr2</b> <b>pwrx</b>	<code>_ARCH_PWR</code> <code>_ARCH_PWR2</code>	pwr2 pwr2s pwrx p2sc	pwr2	yes	no
<b>ppc</b>	<code>_ARCH_PPC</code>	601 602 603 604 403 rs64a rs64b pwr3	604 (32-bit mode) pwr3 (64-bit mode)	yes	yes
<b>ppcgr</b>	<code>_ARCH_PPC</code> <code>_ARCH_PPCGR</code>	603 604	604	yes	yes

-qarch option	Predefined Macro	Available -qtune options	Default -qtune suboption	-q32	-q64
pwr2s	_ARCH_PWR _ARCH_PWR2 _ARCH_PWR2S	pwr2s	pwr2s	yes	no
p2sc	_ARCH_PWR _ARCH_PWR2 _ARCH_P2SC	p2sc	p2sc	yes	no
601	_ARCH_601	601	601	yes	no
602	_ARCH_PPC _ARCH_602	602	602	yes	no
603	_ARCH_PPC _ARCH_PPCGR _ARCH_603	603	603	yes	no
604	_ARCH_PPC _ARCH_PPCGR _ARCH_604	604	604	yes	no
403	_ARCH_PPC _ARCH_403	403	403	yes	no
pwr3	_ARCH_PPC _ARCH_PPCGR _ARCH_PWR3	pwr3	pwr3	yes	yes
rs64a (RS/6000 Models S70, S71)	_ARCH_PPC _ARCH_RS64A	rs64a	rs64a	yes	yes
rs64b (RS/6000 Models S70, S71)	_ARCH_PPC _ARCH_RS64B	rs64b	rs64b	yes	yes
rs64c	_ARCH_PPC _ARCH_RS64C	rs64c	rs64c	yes	yes

### Related Reading

"Invoking the Compiler" on page 8

"Specifying Compiler Options on the Command Line" on page 10

"Specifying Compiler Options in Your Program Source Files" on page 12

"Specifying Compiler Options in a Configuration File" on page 13

"Specifying Compiler Options for Architecture-Specific, 32- or 64-bit Compilation" on page 14

"Appendix G. Built-in Functions for PowerPC Processors" on page 435

"32, 64" on page 231

"arch" on page 237

"tune" on page 331

---

## Compiler Message and Listing Information

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

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

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

If the error is identifiable within the source line, a *finger line* under the source line points to the column position of the error. For example:

```
10 | int add(int, int)
    | ....a...b....c...
a - 1506-166 (S) Definition of function add requires parentheses.
b - 1506-172 (S) Parameter type list for function add contains
    parameters without identifiers.
c - 1506-172 (S) Parameter type list for function add contains
    parameters without identifiers.
```

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

If the **-qlanglvl** option is set to **ansi**, compile-time messages about incorrect **#pragma** directives are not generated.

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
- 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 is produced only when the list option is in effect and 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 Reading

“Compiler Message Format” on page 21

“Message Severity Levels and Compiler Response” on page 20

“w” on page 337

“flag” on page 261

“info” on page 275

“langlvl” on page 286

“source” on page 322

“srcmsg” on page 324

## 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 Reading

“Compiler Message and Listing Information” on page 18

“Compiler Message Format” on page 21

“Compiler Return Codes”

## 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 **E**, **W**, or **I**.
- 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.

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 <b>xlc</b> command cannot allocate any more memory for its use.
251	A signal-received error has been detected. That is, an unrecoverable error or interrupt signal has occurred.
252	A file-not-found error has been detected.
253	An input/output error has been detected: files cannot be read or written to.
254	A fork error has been detected. A new process cannot be created.



**Related Reading**

“Compiler Message and Listing Information” on page 18

“Compiler Message Format”

“Message Severity Levels and Compiler Response” on page 20

“halt” on page 269

“maxerr” on page 299

**Compiler Message Format**

Diagnostic messages have the following format when the “**srcmsg**” on page 324 option is active (which is the default):

*"file", line line\_number.column\_number: 15dd-*nnn*(severity) text.*

where:

<i>file</i>	is the name of the C source file with the error.
<i>line_number</i>	is the line number of the error.
<i>column_number</i>	is the column number for the error
15	is the compiler product identifier
<i>cc</i>	is a two-digit code indicating the C for AIX compiler component that issued the message. <i>cc</i> can have the following values:
<b>00</b>	- code generating or optimizing message
<b>01</b>	- compiler services message.
<b>06</b>	- message specific to C for AIX compiler
<b>40</b>	- message specific to C for AIX compiler
<b>41</b>	- message specific to C for AIX compiler
<b>46</b>	- message specific to C for AIX compiler backend
<b>86</b>	- 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.

To help you find the exact point of the error in the line, when you use the **-qsrcmsg** option, a finger line is produced below the source code line if the error is applicable to a specific column in the source line. For example:

```

10 | int add(int, int)
    | ....a...b....C...
a - 1506-166 (S) Definition of function add requires parentheses.
```

b - 1506-172 (S) Parameter type list for function add contains parameters without identifiers.  
c - 1506-172 (S) Parameter type list for function add contains parameters without identifiers.

The finger line may also be produced in the source listing if you specify the **-qsource** option.

### **Related Reading**

"Compiler Message and Listing Information" on page 18

"Compiler Return Codes" on page 20

"Message Severity Levels and Compiler Response" on page 20

"source" on page 322

"srcmsg" on page 324

---

## Chapter 4. Advanced Compiler Usage

---

### Program Optimization with the C for AIX Compiler

During optimization, the compiler changes the unoptimized code sequences, derived from the source code, into equivalent optimized code sequences. The resulting code runs faster and usually takes less space. However, during optimization, compilation usually takes more time and space.

Because optimization transforms the code, the direct correspondence between source and object code is often lost. Therefore, debugging information is not provided for programs compiled using the optimization option. Optimized code is also more sensitive to subtle coding errors. For these reasons, do not use the optimization options while you are developing your programs. Use the **-O** optimization options only to compile the final versions of your programs.

#### Optimization Levels in C

The default is *not* to optimize your program. To optimize your program, specify one of the following optimizing compiler options:

- **-O**
- **-O2**
- **-O3**
- **-O4**
- **-qOPTimize**
- **-qOPTimize=2**
- **-qOPTimize=3**
- **-qOPTimize=4**

When you specify optimization, the compiler performs a complete control and data-flow analysis for each function. The compiler also uses global register allocation for the whole function, thereby allowing many variables to be kept in registers rather than in memory. The compiler performs optimizations such as described in “Optimization Techniques Used by the C for AIX Compiler”.

#### Related Reading

- “Optimization Techniques Used by the C for AIX Compiler”
- “Special Handling of Math and String Library Functions” on page 25
- “Writing Optimized Program Source Code” on page 197
- “Using Inlined Components” on page 202
- “Minimizing the Size of Object Files” on page 36
- “O, optimize” on page 302
- “Appendix G. Built-in Functions for PowerPC Processors” on page 435

### Optimization Techniques Used by the C for AIX Compiler

Technique	Description of Technique
<b>Value Numbering</b>	Involves constant propagation, expression elimination, and folding of several instructions into a single instruction.
<b>Branch Optimizations</b>	Rearranges the program code to minimize branching logic and to combine physically separate blocks of code.

**Common Subexpression Elimination**

In common expressions, the same value is recalculated in a subsequent expression. The duplicate expression can be eliminated by using the previous value. This step is done even for intermediate expressions within expressions. For example, if your program contains the following statements:

```
a = c + d;  
.  
.  
.  
f = c + d + e;
```

the common expression `c + d` is saved from its first evaluation and is used in the subsequent statement to determine the value of `f`.

**Code Motion**

If variables used in a computation within a loop are not altered within the loop, the calculation can be performed outside of the loop and the results used within the loop.

**Invariant IF Code Floating (Unswitching)**

Removes invariant branching code from loops to make more opportunity for other optimizations.

For example, in the following code segment, the condition test and the conditional assignment:

```
if (a[i]>100.0) b[i]=a[i]-3.7;  
    x+=a[j]+b[i];
```

do not change during execution of the inner loop.

```
for (i=0;i<1000;i++) {  
    for (j=0;j<1000;j++) {  
        if (a[i]>100.0) b[i]=a[i]-3.7;  
        x+=a[j]+b[i];  
    }  
}
```

The compiler translates the code into a machine-language loop that executes as:

```
for (i=0;i<1000;i++) {  
    if (a[i]<100.00) {  
        for (j=0;j<1000;j++) {  
            b[i]=a[i]-3.7;  
            x+=a[j]+b[i];  
        }  
    }  
    else {  
        for (j=0;j<1000;j++) {  
            x+=a[j]+b[i];  
        }  
    }  
}
```

**Reassociation**

Rearranges the sequence of calculations in an array-subscript expression, producing more candidates for common-expression elimination.

**Strength Reduction**

Replaces less efficient instructions with more efficient ones. For example, in array subscripting, an add instruction replaces a multiply instruction.

**Constant Propagation**

Constants used in an expression are combined, and new ones are generated. Some implicit conversions between integer and floating-point types are done.

**Store Motion**

Moves store instructions out of loops.

**Dead Store Elimination**

Eliminates stores when the value stored is never referred to again. For example, if two stores to the same location have no intervening load, the first store is unnecessary and is removed.

**Dead Code Elimination**

Eliminates code that cannot be reached or code whose results are not subsequently used.

**Inlining**

( -Q option )

Replaces function calls with actual program code.

**Instruction Scheduling**

Reorders instructions to minimize execution time.

**Interprocedural Analysis**

( -qipa option )

Uncovers relationships across function calls, and eliminates loads, stores, and computations that cannot be eliminated with more straightforward optimizations.

**Global Register Allocation** Allocates variables and expressions to available hardware registers using a *graph coloring* algorithm.

The **-O** and **-Q** compiler options also determine the types of inlining to be used.

#### Related Reading

“Program Optimization with the C for AIX Compiler” on page 23

“Special Handling of Math and String Library Functions”

“Writing Optimized Program Source Code” on page 197

“O, optimize” on page 302

“Q” on page 314

“ipa” on page 279

## Special Handling of Math and String Library Functions

The C for AIX compiler can improve optimization by generating substitute code for calls to some math and string functions available within the standard C runtime libraries. The functions handled this way are defined as macros in `/usr/include/math.h` or `/usr/include/string.h`.

The special handling of these functions occurs by default, when either `math.h` or `string.h` is included in the source program. To explicitly generate substitute code for a particular function, use the function with two underscores (`__strcpy`, for example).

When including `math.h` and `string.h`, avoid redeclaring the functions.

If your application requires a function call to one or all of the math or string functions, prevent special handling of all math or string functions within a source file by using either the **-U \_\_MATH\_\_** or the **-U \_\_STR\_\_** option on the command line. For example:

```
ixlc -c -U __MATH__ file.c
```

Runtime performance of an application is affected if special handling is disabled.

#### Related Reading

“Program Optimization with the C for AIX Compiler” on page 23

“Optimization Techniques Used by the C for AIX Compiler” on page 23

“U” on page 332

---

## Floating Point Operations with the C for AIX Compiler

### RISC System/6000 Floating Point Hardware

The RISC/6000 floating-point hardware performs all computations in IEEE double precision (eight byte representation), equivalent to **double** in C programs. Single-precision (four byte representation) (**float**) values are automatically converted to double precision before they are used, and all results are calculated in double precision. Double precision provides greater range and precision than single precision does. Double precision values have an approximate range of  $10(-308)$  to  $10(+308)$  and precision of about 16 decimal digits. Single precision values have an approximate range of  $10(-38)$  to  $10(+38)$ , with about 7 decimal digits of precision.

When results must be converted to single precision, rounding operations are used. A rounding operation produces the correct single-precision value based on the IEEE rounding mode in effect. Because explicit rounding operations are required, single-precision computations are often slower than double precision computations. On many other machines the reverse is true: single-precision operations are faster than

double-precision operations. Code ported from other systems can show different performance on a RISC System/6000 computer. See the **-qfloat=rndsngl** compiler option for more information about single precision.

The RISC System/6000 hardware also provides a special set of double-precision operations that multiply two numbers and add a third number to the product. These combined multiply-add (**maf**) operations are performed in the same time as a multiply or an add operation alone. The **maf** functions provide an extension to the IEEE standard because they perform the multiply and add with one (rather than two) rounding errors. The **maf** functions are both faster and more accurate than the equivalent separate operations. Use the **nomaf** option to suppress the generation of these multiply-add instructions.

**Note:** PowerPC and Power3 hardware can perform computations in either *single* or *double* precision. Considerations regarding single precision do not apply to these platforms.

## Detecting Floating-Point Exceptions

A number of floating-point exceptions can be detected by the floating-point hardware: invalid operation, division by zero, overflow, underflow, and inexact. By default, all exceptions are ignored. However, if you use the **fltrap** option, any or all of these exceptions can be detected. (For an example of how this works, see “Sample TRAP Signal Handler” on page 31.) In addition, when you add suitable support code to your program, program execution can continue after an exception occurs, and you can then modify the results of operations causing exceptions.

Refer to “Floating-Point Processor Overview” and “Floating-Point Exceptions” in the *AIX Version 4 Assembler Language Reference* for more information about RISC System/6000 floating-point processing.

### Related Reading

“Compile-Time Floating-Point Arithmetic”  
“Floating-Point Compiler Options” on page 27  
“Rounding Mode Restrictions” on page 35  
“Sample TRAP Signal Handler” on page 31  
“float” on page 261  
“fltrap” on page 264

## Compile-Time Floating-Point Arithmetic

The compiler attempts to perform as much floating-point arithmetic as possible at compile time. Floating-point operations with constant operands are folded, replacing the operation with the result calculated at compile time. When the **-O** option is used, more folding might occur than when optimization is not enabled.

All compile-time folding of floating-point computations can be suppressed using the **float=nofold** option. Alternatively, the IEEE rounding mode used in compile-time arithmetic can be controlled using the **-y** 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 **maf** 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 runtime operation. This would prevent an exception from occurring at run time. When using the **fltrap** option, you should consider using the **float=nofold** option.

In general, code that affects the rounding mode at run time should be compiled with the **-y** option that matches the rounding mode intended at run time. For example, when the following program:

```
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);
}
```

is compiled with:

```
xlc -yz -qfloat=rndsngl
```

the expression **1.0/3.0** is folded by the compiler at compile time into a double-precision result. This result is then converted to single precision and then stored in **float x**. The **float=nofold** option can be specified to suppress all compile-time folding of floating-point computations. The **-yz** option only affects compile-time rounding of floating-point computations, but does *not* affect runtime rounding. The code fragment:

```
x = 1.0;
y = 3.0;
x = x/y;
```

is evaluated at run time in single precision. Here, the default runtime rounding of “round to nearest” is still in effect and takes precedence over the compile-time specification of “round to zero” (**-yz**).

**Note:** The **-y** option *does not* specify the runtime rounding mode.

### Related Reading

“Floating-Point Compiler Options”

“Rounding Mode Restrictions” on page 35

“O, optimize” on page 302

“y” on page 339

“float” on page 261

“flttrap” on page 264

“maf” on page 297

## Floating-Point Compiler Options

Compiler options affect the accuracy, performance, and potentially the correctness of floating-point calculations. Although the default values for the options have been chosen to provide efficient and correct execution of most programs, some applications may require nondefault options to reproduce results reported by other hardware. You should read this and related pages before using the floating-point options.

By default, the C for AIX compiler produces object code that evaluates floating-point expressions in double precision, even if all operands in an expression are single precision. The results of expressions are then rounded to single precision if they are assigned to **float** variables.

Other C compilers might evaluate floating-point expressions in single precision where such an evaluation is permitted by the language definition. This implementation is preferred on machines where single-precision operations are faster than double-precision operations.

In general, floating-point results from programs compiled using C for AIX and executed on the RISC System/6000 system are more accurate than those from other implementations, because of the higher

precision used for intermediate results and the use of **maf** operations.

### Related Reading

“float” on page 261 Compiler Option  
“flttrap” on page 264 Compiler Option  
“-qfloat=nomaf”  
“-qfloat=hssngl”  
“-qfloat=nans” on page 29  
“-qfloat=hsflt” on page 29  
“-qfloat=rndsngl” on page 30  
“-qflttrap” on page 30  
“Sample TRAP Signal Handler” on page 31

## **-qfloat=nomaf**

The **nomaf** option is provided for cases where it is necessary to exactly duplicate the **double** results of an implementation that does not have multiply-add operations. The **nomaf** option prevents the compiler from generating any multiply-add operations. Not using multiply-add operations decreases accuracy and performance but strictly conforms to the IEEE standard for double-precision arithmetic.

To duplicate the single-precision results from other implementations, you may also need to use the **-qfloat=rndsngl** option.

### Related Reading

“Floating-Point Compiler Options” on page 27  
“-qfloat=hssngl”  
“-qfloat=nans” on page 29  
“-qfloat=hsflt” on page 29  
“-qfloat=rndsngl” on page 30  
“-qflttrap” on page 30  
“Sample TRAP Signal Handler” on page 31  
“float” on page 261  
“flttrap” on page 264

## **-qfloat=hssngl**

The **-qfloat=hssngl** option improves the performance of single-precision (**float**) floating-point calculations by suppressing certain rounding operations. The suppressed rounding operations are required by the C language, but are not necessary for correct program execution. Rounding operations are still inserted when double-precision results are stored into single-precision memory locations.

The **hssngl** option retains the results of floating-point expressions in double precision when the original program would have rounded those results to single precision. The retained double-precision results are then used in later expressions instead of the rounded results. The program results may be more accurate because of the increased precision, and program execution may be faster because rounding operations have been omitted.

Rounding operations are still necessary in cases where a floating-point result is stored into a single-precision variable. The result must be rounded to detect a single-precision floating-point overflow or underflow. In some cases, program optimization can remove store operations from a program. The **hssngl** option allows the rounding operation that accompanied the original store to be removed also. When the **hssngl** option can retain such a result in double precision, an exception can be avoided.

The **hssngl** option is safe for all types of programs because it can only increase the precision of floating-point computations in a program.



Use the **-qfloat=hssngl** option with the **-O** option, but not with the **-qfloat=rndsngl** option.

#### Related Reading

“Floating-Point Compiler Options” on page 27

“-qfloat=nomaf” on page 28

“-qfloat=nans”

“-qfloat=hsflt”

“-qfloat=rndsngl” on page 30

“-qflttrap” on page 30

“Sample TRAP Signal Handler” on page 31

“O, optimize” on page 302

“float” on page 261

“flttrap” on page 264

### **-qfloat=nans**

The **-qfloat=nans** option causes the compiler to generate object code that detects the conversion of a single-precision signalling NaN (NaNs) to double precision. By default, the compiler generates object code that detects the use of a NaNs in all other situations required by the IEEE standard.

Very few programs actually require checks for NaNs. A NaNs cannot be produced by any floating-point operation, and must instead be explicitly created. A program only needs to be compiled with the **-qfloat=nans** option if it explicitly creates a signalling NaN.

#### Related Reading

“Floating-Point Compiler Options” on page 27

“-qfloat=nomaf” on page 28

“-qfloat=hssngl” on page 28

“-qfloat=hsflt”

“-qfloat=rndsngl” on page 30

“-qflttrap” on page 30

“Sample TRAP Signal Handler” on page 31

“float” on page 261

“flttrap” on page 264

### **-qfloat=hsflt**

The **-qfloat=hsflt** option improves the performance of floating-point computations by suppressing all rounding operations and by performing conversions from floating point to integer with inline code. This option is intended for knowledgeable programmers in specific applications where the computational characteristics of a program are known. To safely use the **hsflt** option, a program must *never* attempt to assign floating-point results to single-precision variables unless the results are known to be within the allowable range of single-precision values. In addition, if any floating-point numbers are converted to integers, the floating-point numbers must be within the representable range of integers. If the **hsflt** option is used in cases where a program does not have these properties, the program may produce incorrect results without warning. When the computational characteristics of a program are not known, use **hssngl** not **hsflt**.

In suppressing rounding operations, the **hsflt** option operates in the same way as the **hssngl** option. However, the **hsflt** option also suppresses rounding operations when double-precision values are assigned to single-precision variables. Single-precision overflow or underflow is not detected in such assignments, and the assigned value is not properly rounded according to the current rounding mode.

For floating-point-to-integer conversions, the **hsflt** option allows the compiler to use inline code sequences instead of subroutine calls. The inline code sequences do not check the floating-point value, and produce incorrect results in cases where the floating-point value does not fall within the range of an integer.

Use the **-qfloat=hsflt** option with the **-O** option, but not with the **-qfloat=rndsngl** or **-qfloat=hssngl** options.

#### Related Reading

“Floating-Point Compiler Options” on page 27

“-qfloat=nomaf” on page 28

“-qfloat=hssngl” on page 28

“-qfloat=nans” on page 29

“-qfloat=rndsngl”

“-qflttrap”

“Sample TRAP Signal Handler” on page 31

“O, optimize” on page 302

“float” on page 261

“flttrap” on page 264

### **-qfloat=rndsngl**

The **-qfloat=rndsngl** option is provided for cases where it is necessary to exactly duplicate the results of an implementation that uses single-precision floating-point arithmetic for **float** expressions. The **rndsngl** option causes the compiler to round the results of floating-point operations on **float** operands to single precision. The effect of rounding the intermediate results to single precision is the same as if single-precision operations had been used for evaluating **float** expressions. Runtime performance can decrease significantly because of the increased rounding overhead.

Some programs might check portions of their results by comparing those results with values computed on other systems. Again, the **rndsngl** option may be required to duplicate the previous results and to have such programs report correct execution. Programs checking double-precision results may also require the **nomaf** option.

#### Related Reading

“Floating-Point Compiler Options” on page 27

“-qfloat=nomaf” on page 28

“-qfloat=hssngl” on page 28

“-qfloat=nans” on page 29

“-qfloat=hsflt” on page 29

“-qflttrap”

“Sample TRAP Signal Handler” on page 31

“float” on page 261

“flttrap” on page 264

### **-qflttrap**

The IEEE standard for floating-point arithmetic specifies that five types of exceptions be signalled when detected:

- overflow
- underflow
- division by zero
- invalid operation
- inexact

By default, the signalling of an exception involves setting a status flag and continuing. The standard also allows for an exception to generate a trap and invoke a handler routine specified by the user.

The **flttrap** option directs the compiler to produce code that generates a **TRAP** signal to flag the occurrence of any enabled floating-point exception. Exception types can be specified with the **flttrap** option. Each of the five exception types is controlled by a separate suboption:

<b>Overflow</b>	Generates code to detect and trap floating-point overflow.
<b>UNderflow</b>	Generates code to detect and trap floating-point underflow.
<b>ZERODivide</b>	Generates code to detect and trap floating-point division by zero.
<b>INValid</b>	Generates code to detect and trap floating-point invalid-operation exceptions.
<b>INEXact</b>	Generates code to detect and trap floating-point inexact exceptions.

The exceptions are enabled using the **-qfltrap=enable** option or the Base Operating System (BOS) Runtime Service routine **fp\_enable**. The **enable** suboption inserts code into the prologue of the main program to enable the exceptions specified by the **-qfltrap** option. The suboption has no effect on source files that do not contain a main program.

The **-qfltrap=imprecise** suboption generates code that checks for the specified exceptions only on entry and exit to functions that perform floating-point computations. If an exception occurs, it is detected, but the exact location of the exception is not determined. When the **imprecise** suboption is not specified, each floating-point operation in the code compiled with the **-qfltrap** option is checked. Unless the exception occurred during a call to another function that was not compiled with **-qfltrap** (for example, a library routine), the exact location of any exception is identified.

Specifying the **-qfltrap** option with no suboptions is equivalent to setting

```
-qfltrap=ov:und:zero:inv:inex
```

The exceptions are not automatically enabled, and all floating-point operations are checked to provide precise exception-location information.

By default, the **TRAP** signals generated by enabled exceptions cause a program to stop. Alternatively, the exceptions can be acted upon by a program by providing a routine that is to be invoked when a **TRAP** signal occurs, and by calling the BOS Runtime Service routine to specify that routine as the handler of **TRAP** signals. In these respects, the implementation of **-qfltrap** does not fully support the exception-handling environment suggested by the IEEE floating-point standard.

Floating-point exceptions are described in the *AIX Version 4 Assembler Language Reference*. The sample **“Sample TRAP Signal Handler”** signal handler illustrates the detection and handling of floating-point exceptions.

#### Related Reading

“Floating-Point Compiler Options” on page 27

“-qfloat=nomaf” on page 28

“-qfloat=hssngl” on page 28

“-qfloat=nans” on page 29

“-qfloat=hsflt” on page 29

“-qfloat=rndsngl” on page 30

“Sample TRAP Signal Handler”

“float” on page 261

“fltrap” on page 264

**Sample TRAP Signal Handler:** The sample C code below defines a **TRAP** signal handler `fhandler_`. It uses the **fp\_enable** and **fp\_disable\_all** support routines from the Base Operating System (BOS) Runtime Services to enable or disable floating-point exceptions. The sample handler prints an error message indicating the type and location of the operation that caused the exception. You can use a load map and compiler listing to show the location and identify the source code line that generated the exception. The signal-handling code also allows the results of failing instructions to be modified to specific values.

The program `myprogram.c` would be compiled with the command:

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

and the resulting object file would be linked with other C object files produced using the **fltrap** option.

**Note:** This code is for illustrative purposes; even when support code such as this is used, the implementation of **fltrap** does not fully support the exception-handling environment suggested by the IEEE floating-point standard.

```

/*
 * Exception handling support for use with the 'fltrap' compiler
 * option. Provides routines to enable, disable, and handle
 * exceptions. Exception handling includes the ability to
 * identify the point where an exception occurred and to continue
 * execution following an exception, possibly supplying a value
 * as the result of the failing instruction.
 *
 * Two routines are visible:
 *   enable_fp_traps_(mask)
 *   disable_fp_traps_()
 * The names contain a trailing underscore to enable their use
 * with the FORTRAN 'extname' compiler option.
 *
 * The fltrap compiler option will generate TRAP signals when
 * floating-point exceptions occur. It does so by setting the
 * record bit on all floating-point instructions, and then
 * trapping if condition register bit 5 is set (that is, if
 * the floating-point enabled exception (FEX) bit is set in
 * the floating-point status and control register).
 */
#include <stdio.h>   #include <stdlib.h>   #include <signal.h>   #include <fptrap.h>   #include <fpxcp.h>   /*
 * The specific trap instruction used by the fltrap option is
 * TRAP R15=R15. This is the machine code for that instruction.
 */
#define FLTRAPINST (0x7c8f7808)
/*
 * The following table maps instruction bit patterns to the name
 * of a floating-point instruction. This table is referenced
 * using bits 26-30 of a floating-point instruction.
 */
static char *op_table[32] = {
    "fcmp", "?", "?", "?", "?", "?", "?", "?",
    "?", "?", "?", "?", "frsp", "?", "?", "?",
    "?", "?", "fd", "?", "fs", "fa", "?", "?",
    "?", "fm", "?", "?", "fms", "fma", "fnms", "fnma" };
/*
 * The following variables record the location of the failing
 * operation, the kind of operation, and the floating-point
 * registers found in a failing instruction. Note that the
 * valid registers depend on the instruction type.
 */
static unsigned int *fpe_loc;
static char *opcode;
static int frt_reg, fra_reg, frb_reg, frc_reg;
/* Mask value to check for floating-point exceptions. */
#define TST_MASK (FP_INVALID|FP_OVERFLOW| \
    FP_UNDERFLOW|FP_DIV_BY_ZERO|FP_INEXACT)
/* Function Prototypes */
static int find_instr(unsigned int *trap_loc);
void enable_fp_traps_(int *mask);
void disable_fp_traps_();
/*
 * Sample exception handler.
 * Customize this code by printing additional debugging
 * information and defining exception results.
 */
static void fhandler_(int sig,int code,struct sigcontext *scp)
{
    fptrap_t fpstat;
    int result_reg;

```

```

    fpstat = scp->sc_jmpbuf.jmp_context.fpscr;
    /* Check that the trap is of the type used for the flttrap
     * option and that the floating-point status and control
     * register indicates that an exception has occurred.
     */
    if (*(int *) scp->sc_jmpbuf.jmp_context.iar) != FLTTRAPINST ||
!(fpstat & TST_MASK) {
/*
 * This must be a trap caused by an integer division by
 * zero or a subscript out of range. */
fputs("SIGTRAP without floating-point exception\n",stderr);
exit(42);
}
/*
 * Find the floating-point instruction causing the exception and
 * decode it. find_inst sets the static variables that indicate
 * the instruction location, kind, and registers.
 */
    if (find_instr((unsigned int *)scp->sc_jmpbuf.jmp_context.iar)) {
fputs("SIGTRAP handler failed to find exception point\n",
    stderr);
/*
 * Note that, because the exception might have occurred in a
 * subroutine that was not compiled with the flttrap option,
 * it may be desirable simply to ignore the exception by
 * clearing the exception bits and returning.
 */
exit(43);
}
/* Examine the floating-point status and control register for
 * enabled exceptions. Customize each case below. */
    if ((fpstat & (TRP_INVALID|FP_INVALID)) ==
(TRP_INVALID|FP_INVALID)) {
fprintf(stderr,
    "FP invalid operation, operation '%s', location %x\n",
    opcode, fpe_loc);
/*
 * Consider an invalid operation an unrecoverable error.
 * By examining other bits in the status and control register,
 * we can identify the specific invalid operation that
 * occurred (for example, zero divided by zero). Using the
 * kind of operation, we can examine the source operands.
 * If the instruction has any result registers, they
 * have not been modified.
 */
exit(44);
}
    if ((fpstat & (TRP_OVERFLOW|FP_OVERFLOW)) ==
(TRP_OVERFLOW|FP_OVERFLOW)) {
fprintf(stderr,"FP overflow, operation '%s', location %x\n",
    opcode, fpe_loc);
/*
 * Note that the result register in an overflow contains a
 * correctly rounded normalized number, but 1536 has been
 * subtracted from the exponent.
 * Set the result of any overflow to zero.
 */
scp->sc_jmpbuf.jmp_context.fpr[frr_reg] = 0.0;
}
    if ((fpstat & (TRP_UNDERFLOW|FP_UNDERFLOW)) ==
(TRP_UNDERFLOW|FP_UNDERFLOW)) {
fprintf(stderr,"FP underflow, operation '%s', location %x\n",
    opcode, fpe_loc);
/*
 * Note that the result register in an underflow contains a
 * correctly rounded normalized number, but 1536 has been
 * added to the exponent.

```

```

    * Set the result of any underflow to zero.
    */
    scp->sc_jmpbuf.jump_context.fpr[fpr_reg] = 0.0;
}
    if ((fpstat & (TRP_DIV_BY_ZERO|FP_DIV_BY_ZERO)) ==
        (TRP_DIV_BY_ZERO|FP_DIV_BY_ZERO)) {
    fprintf(stderr,
        "FP division by zero, operation '%s', location %x\n",
        opcode, fpe_loc);
    /*
    * Print the source operands for the division; the divide
    * instruction uses FRA and FRB. Note that the result
    * register has not been modified by the divide.
    */
    fprintf(stderr, " Division source operands: %f / %f\n",
        scp->sc_jmpbuf.jump_context.fpr[fra_reg],
        scp->sc_jmpbuf.jump_context.fpr[frb_reg]);
    /* Set the result of any division by zero to zero.    */
    scp->sc_jmpbuf.jump_context.fpr[fpr_reg] = 0.0;
}
    if ((fpstat & (TRP_INEXACT|FP_INEXACT)) ==
        (TRP_INEXACT|FP_INEXACT)) {
    fprintf(stderr, "FP inexact, operation '%s', location %x\n",
        opcode, fpe_loc);
    /* No action, just ignore this. */
}
    /* Reset the exception bits because they are sticky. */
    scp->sc_jmpbuf.jump_context.fpscr &= ~FP_ALL_XCP;
    /* signal(SIGTRAP, fhandler_); */
    /* Continue execution with the instruction following the trap.*/
    scp->sc_jmpbuf.jump_context.iar += 4;
}
/*
* Find and decode the floating-point instruction causing the
* exception. Return 1 if not found, else zero.
*/
static int find_instr(unsigned int *trap_loc)
{
    /*
    * Search backward in the instruction stream starting from
    * trap_loc, looking for a floating-point instruction (bits
    * 0-5 equal decimal 63). The first such instruction found
    * will be assumed to be the failing operation.
    * Note that a linear backward search assumes that there is
    * no branching code separating the trap instruction from
    * the failing floating-point operation. This will always
    * be true with the current implementation of the flttrap
    * option (in fact, in the current implementation the
    * failing operation will always be the second last
    * instruction before the trap point), except in the case
    * of subroutine calls causing an exception.
    * For safety we limit the search length.
    */
    int i = 0;
    while ((*(-trap_loc) >> 26 != 63) &&
        (++i < 10));
    if (*trap_loc >> 26 != 63) return(1); /* no float op found */
    /* Check that the operation found has the record bit set. */
    if (!( *trap_loc & 1)) return(1); /* record bit not set */
    /*
    * Check to see if the instruction found was a move register.
    * This instruction is produced after calls to external
    * routines to see if they returned with any exception bits
    * set. Any such external routine must be a library routine
    * or in user code that was not compiled with flttrap.
    */
    if ((( *trap_loc >> 1) & 0x3ff) == 72) return(1); /* fmr found */
}

```

```

/* Decode the instruction to identify the kind of operation
 * and the source and result registers. */
fpe_loc = trap_loc;
opcode = op_table[(*trap_loc >> 1) & 0x1f];
frt_reg = (*trap_loc >> 21) & 0x1f;
fra_reg = (*trap_loc >> 16) & 0x1f;
frb_reg = (*trap_loc >> 11) & 0x1f;
frc_reg = (*trap_loc >> 6) & 0x1f;
return(0);
}
/*
 * Install a trap handler and enable floating-point exceptions.
 * The mask parameter indicates which exceptions should be enabled
 * as follows (values are from /usr/include/fptrap.h):
 *   Invalid Operation = TRP_INVALID = 0x00000080
 *   Overflow = TRP_OVERFLOW = 0x00000040
 *   Underflow = TRP_UNDERFLOW = 0x00000020
 *   Division by Zero = TRP_DIV_BY_ZERO = 0x00000010
 *   Inexact = TRP_INEXACT = 0x00000008
 * To enable multiple exceptions, OR values together. Note that
 * the parameter is a pointer, for FORTRAN call by reference.
 */
void enable_fp_traps_(int *mask)
{
    signal(SIGTRAP,(void(*)())fhandler_);
    fp_enable(*mask);
}
/* Disable all floating-point exceptions and remove trap handler.*/
void disable_fp_traps_()
{
    fp_disable_all();
    signal(SIGTRAP,SIG_DFL);
}

```

## Rounding Mode Restrictions

The floating-point rounding mode can only be changed at the beginning and end of a function. It cannot be changed across a function call, and if it is changed within a function, it must be restored before returning to the calling routine.

### Related Reading

“Floating-Point Compiler Options” on page 27

“Compile-Time Floating-Point Arithmetic” on page 26

---

## Creating and Using Precompiled Headers

You can improve your compile time by using precompiled headers. Use the **-qgenpcomp** and **-qusepcomp** compiler options to create and maintain precompiled header files for your application.

If you use these two options consistently, a precompiled header file is created if it does not exist, and used if it does exist. When a source file is changed, the precompiled version is automatically regenerated.

The compiler generates a single precompiled object for the first initial sequence of **#include** directives. The next time you compile, this single object can be used wherever that initial sequence appears. Since the precompiled object is only used in cases where the context is the same (same language, same beginning sequence of **#include** directives, compatible options and macro definitions), the precompiled object does not have to be reinterpreted every time it is included.

To get the most benefit from this new method, use the same initial sequence of headers wherever possible. The more files that share the same initial sequence, the greater the improvement in your compile time.

You can specify different names or directories for precompiled header files with the **-qgenpcomp** and **-qusepcomp** compiler options. This allows you to create more than one initial sequence, and further improve your compile time.

When you use precompiled header files, the following restrictions apply:

- To create a precompiled header file, the compiler process must have write permission to the directories you specify, or to the current working directories if none are specified.
- To use a precompiled header, the compiler process must have read permission for that file.
- Precompiled header files do not appear in any listing files.

#### **Related Reading**

"#include Preprocessor Directive" on page 356

"genpcomp" on page 267

"usepcomp" on page 335

---

## **Minimizing the Size of Object Files**

To minimize the size of object files, you can specify the **-qcompact** option. Using this option may increase execution time.

#### **Related Reading**

"compact" on page 246



---

## Chapter 5. Program Parallelization

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

- Automatic and explicit parallelization of countable loops using IBM pragma directives.
- Program parallelization 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 run-time options and calls to library functions. Work is distributed among available threads according to the directives specified in the source.

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

---

### IBM Directives

IBM 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 its **omp** suboption. The compiler must be invoked using a thread-safe compiler mode.

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

The C for AIX 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.

---

### 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 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 Reading

“Countable Loops”  
 “Reduction Operations in Parallelized Loops” on page 39  
 “Shared and Private Variables in a Parallel Environment” on page 40  
 “Compiler Modes” on page 5  
 “Using Pragmas to Control Parallel Processing” on page 41  
 “Invoking the Compiler” on page 8  
 “#pragma Preprocessor Directives for Parallel Processing” on page 381  
 “Run-time Options for Parallel Processing” on page 402  
 “OpenMP Run-time Options for Parallel Processing” on page 404  
 “Built-in Functions Used for Parallel Processing” on page 400  
 “smp” on page 320  
 “strict” on page 326  
 “strict\_induction” on page 327

---

## Countable Loops

A loop is considered to be *countable* if :

- there is no branching into or outside 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:

<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>

- 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 compile-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 Reading

- “Chapter 5. Program Parallelization” on page 37
- “Shared and Private Variables in a Parallel Environment” on page 40
- “Reduction Operations in Parallelized Loops”
- “Using Pragmas to Control Parallel Processing” on page 41
- “#pragma Preprocessor Directives for Parallel Processing” on page 381

---

## 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:

*var* 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:

```
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;
    }
}
```

*op* Is one of the following operators:

```
+ - * ^ | &
```

*assign\_op* Is one of the following operators:

```
+= -= *= ^= |= &=
```

*expr*

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.

### Related Reading

"Chapter 5. Program Parallelization" on page 37

"Countable Loops" on page 38

"Shared and Private Variables in a Parallel Environment"

"Using Pragmas to Control Parallel Processing" on page 41

"#pragma Preprocessor Directives for Parallel Processing" on page 381

"#pragma ibm critical Preprocessor Directive" on page 382

"info" on page 275

---

## 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 constructs.
- 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.

You can explicitly specify a shared or private context for a variable, or you can let the compiler determine the default context of a variable according to the following rules:

- Variables with **static** storage duration are shared.
- Dynamically allocated objects are shared.
- Variables with automatic storage duration are private.
- All variables defined outside a parallel construct become shared when the parallel construct 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 by the **alloca** function within:
  - a parallel loop or any other OpenMP construct persists only for the duration of that construct and is private for each thread.
  - a work-sharing loop persists only for the duration of one iteration of that loop.
  - a section of a work-sharing sections construct persists only for the duration of that section.

The following code segments show examples of these 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;...) {
      int tmp = b;                       /* b is still private here ! */
      foo (i);                           /* i is private here because it */
                                        /* is an iteration variable */
    }

    #pragma omp parallel
    {
      int tmp = 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 */
  ... }

```

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

Note that even if a variable has shared context, it can be privatized by the compiler 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 variables are reported by the **-qinfo=private** option.

#### Related Reading

“Chapter 5. Program Parallelization” on page 37

“Countable Loops” on page 38

“Reduction Operations in Parallelized Loops” on page 39

“Using Pragmas to Control Parallel Processing”

“#pragma Preprocessor Directives for Parallel Processing” on page 381

“#pragma ibm critical Preprocessor Directive” on page 382

“info” on page 275

---

## Using Pragmas to Control Parallel Processing

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

### IBM Directives

**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

**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 parallel region {
  /* code here is executed by all threads */
  #pragma omp sections {
    /* each section is executed once */
    #pragma omp section
    structured_block_1
  }
}
```

```
    ...  
    #pragma omp section  
    structured_block_2  
    ...  
    ....  
  }  
}
```

### **Related Reading**

“Chapter 5. Program Parallelization” on page 37

“Shared and Private Variables in a Parallel Environment” on page 40

“Countable Loops” on page 38

“#pragma Preprocessor Directives for Parallel Processing” on page 381

“smp” on page 320

“info” on page 275





---

## Chapter 6. The C Language

C is a programming language designed for a wide variety of programming tasks. It is used for system-level code, text processing, graphics, and in many other application areas.

The C language described in these pages is consistent with the Systems Application Architecture Common Programming Interface (also known as the SAA C Level 2 interface), and with the International Standard C (ANSI/ISO-IEC 9899-1990[1992]). This standard has officially replaced American National Standard for Information Systems—Programming Language C (X3.159-1989) (X3.159-1989) and is technically equivalent to the ANSI\*\* C standard.

C supports several data types, including characters, integers, floating-point numbers, and pointers — each in a variety of forms. In addition, C also supports arrays, structures (records), unions, and enumerations.

The C language contains a concise set of statements, with functionality added through its library. This division enables C to be both flexible and efficient. An additional benefit is that the language is highly consistent across different systems.

The C library contains functions for input and output, mathematics, exception handling, string and character manipulation, dynamic memory management, as well as date and time manipulation. Use of this library helps to maintain program portability, because the underlying implementation details for the various operations need not concern the programmer.

All of the standard C library functions and many others are part of the AIX Base Operating System (BOS) Runtime Services. The *AIX Version 4 Technical Reference, Volumes 1 and 2: Base Operating System and Extensions* describes all of the C library functions supported by the C for AIX compiler. Refer to “Subroutines Overview” in *AIX Version 4 System User’s Guide: Operating System and Devices* for general information about library functions.

---

## Lexical Elements of C

### Tokens

Source code is treated during preprocessing and compilation as a sequence of tokens. There are five different types of tokens:

- Identifiers
- Keywords
- Literals
- Operators
- Other separators

Adjacent identifiers, keywords and literals must be separated with white space. Other tokens should be separated by white space to make the source code more readable. White space includes blanks, horizontal and vertical tabs, new lines, form feeds and comments.

### Related Reading

“Comments” on page 46

“Identifiers” on page 47

“Constants” on page 48

## Comments

Comments begin with the `/*` characters, end with the `*/` characters, and can span more than one line. You can put comments anywhere the language allows white space. Multibyte characters can be included in a comment.

Comments are replaced during preprocessing by a single space character.

If the “**cplusplus**” on page 247 compiler option is in effect when you compile a C program, double slashes (`//`) also specify the beginning of a comment. The comment ends at the next newline character.

The “**C**” on page 242, “**E**” on page 253, and “**P**” on page 307 compiler options affect how comments appear in the compiler listing.

**Note:** The `/*` or `*/` characters found in a character constant or string literal do not start or end comments.

You cannot nest comments. Each comment ends at the first occurrence of `*/`. For example, in the following code segment, the comments are highlighted:

```
1  /* A program with nested comments. */ 2
3  #include <stdio.h>
4
5  int main(void)
6  {
7      test_function();
8  }
9
10 int test_function(void)
11 {
12     int number;
13     char letter;
14     /*15  number = 55;16  letter = 'A';17  /* number = 44; */18  */
19     return 999;
20 }
```

In `test_function`, the compiler reads the `/*` in line 14 through the `*/` in line 17 as a comment, and line 18 as C language code, causing errors at line 18.

To avoid commenting over comments already in the source code, you can use conditional compilation preprocessor directives to cause the compiler to bypass sections of a C program. For example, one method to ignore lines 15 through 17 would be to change line 14 to:

```
14  #if 0
```

and line 18 to:

```
18  #endif
```

To later reenable the ignored comments, change line 14 to:

```
14  #if 1
```

Conditional compilation preprocessor directives are described in Preprocessor Directives.

### Related Reading

“Tokens” on page 45

“Identifiers”

“Constants” on page 48

“Preprocessor Directives” on page 58

“Conditional Compilation Directives” on page 60

“#if, #elif Preprocessor Directives” on page 352

“#endif Preprocessor Directive” on page 353

## Identifiers

Identifiers consist of an arbitrary number of letters or digits. They provide names for the following language elements:

- Functions
- Data objects
- Labels
- Tags
- Parameters
- Macros
- Typedefs
- Structure and union members.

There is no limit for the number of characters in an identifier. However, the linkage editor does limit the number of significant characters in external identifiers and truncates them after 4095 characters.

The compiler distinguishes between uppercase and lowercase letters in identifiers. For example, `PROFIT` and `profit` represent different data objects.

**Note:**The underscore character ( `_` ) is considered a letter. In **ansi** mode, identifiers can begin with `_` but not with `$`. In **extended** mode, identifiers can begin with `_` or `$`, but you should avoid using these characters at the beginning of identifiers because they are reserved for internal system names. . The “**dollar**” on page 253 compiler option lets you use the `$` character in identifiers.

Identifiers used by C library functions that begin with two underscores or an underscore followed by a capital letter, are reserved in all contexts.

Although the names of system calls and library functions are not reserved words if you do not include the appropriate headers, avoid using them as identifiers. Duplication of a predefined name can lead to confusion for the maintainers of your code and can cause errors at link time or run time. If you include a library in a program, be aware of the function names in that library to avoid name duplications.

You should always include the appropriate headers when using standard library functions.

### Related Reading

“Tokens” on page 45

“Comments” on page 46

“Constants” on page 48

“Scope of Identifier Visibility” on page 53

## Constants

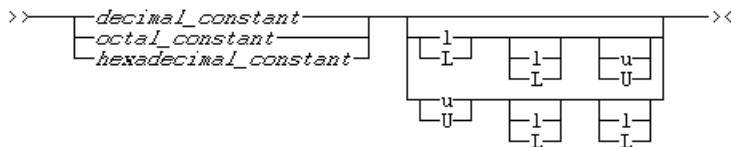
A constant does not change its value while the program is running. The value of any constant must be in the range of representable values for its type.

The C language contains the following types of constants (also called *literals*):

- “Integer Constant” on page 47
- “Floating-Point Constants” on page 49
- “Character Constants” on page 50
- “String Literals” on page 51
- “Escape Sequences” on page 52
- “enum” on page 82

### Integer Constant

*Integer constants* can represent decimal, octal, or hexadecimal values.



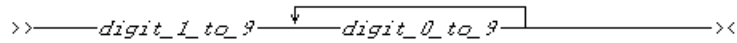
**Data Types for Integer Constants:** The data type of an integer constant is determined by the form, value, and suffix of the constant. The following table lists the integer constants and shows the possible data types for each constant. The smallest data type that can represent the constant value is used to store the constant.

Assigned Constant Value	Data Types for Integer Constants
unsuffixed decimal	int, long int, unsigned long int
unsuffixed octal	int, unsigned int, long int, unsigned long int
unsuffixed hexadecimal	int, unsigned int, long int, unsigned long int
suffixed by <b>u</b> or <b>U</b>	unsigned int, unsigned long int
suffixed by <b>l</b> or <b>L</b>	long int, unsigned long int
suffixed by both <b>u</b> or <b>U</b> , and <b>l</b> or <b>L</b>	unsigned long int
suffixed by <b>ll</b> or <b>LL</b>	long long int, unsigned long long int
suffixed by both <b>u</b> or <b>U</b> , and <b>ll</b> or <b>LL</b>	unsigned long long int

A plus (+) or minus (-) symbol can precede the constant. It is treated as a unary operator rather than as part of the constant value.

Note that the integer constant -2147483648 is not valid because 2147483648 is an **unsigned int** value, which cannot have the unary minus operator applied to it. Instead, this value should be coded as -(2147483647 + 1). To avoid such problems with very small integral values, you should use the identifiers **INT\_MIN** (for **int**), **SHRT\_MIN** (for **short int**), and **SCHAR\_MIN** (for **signed char**). These and other limits for integer values are set in the `/usr/include/limits.h` include file. Header files are described in the *AIX Version 4 Files Reference*.

**Decimal Values:** A *decimal constant* contains any of the digits 0 through 9. The first digit cannot be 0.

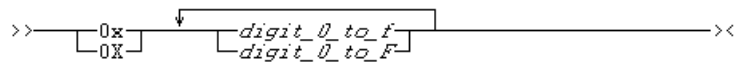


Integer constants beginning with the digit 0 are interpreted as an octal constant, rather than as a decimal constant.

The following are examples of decimal constants:

```
485976
-433132211
+20
5
```

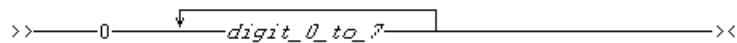
**Hexadecimal Values:** A hexadecimal constant begins with the 0 digit followed by either an x or X, followed by any combination of the digits 0 through 9 and the letters a through f or A through F. The letters A (or a) through F (or f) represent the values 10 through 15, respectively.



The following are examples of hexadecimal constants:

```
0x3b24
0XF96
0x21
0x3AA
0X29b
0X4bD
```

**Octal Values:** An octal constant begins with the digit 0 and contains any of the digits 0 through 7.



The following are examples of octal constants:

```
0
0125
034673
03245
```

## Floating-Point Constants

A floating-point constant consists of:

- An integral part
- A decimal point
- A fractional part
- An exponent part
- An optional suffix.

Both the integral and fractional parts are made up of decimal digits. You can omit either the integral part or the fractional part, but not both. You can omit either the decimal point or the exponent part, but not both.

A suffix of **f** or **F** indicates a type of **float**, and a suffix of **l** or **L** indicates a type of **long double**. If a suffix is not specified, the floating-point constant has a type **double**.

A plus (+) or minus (-) symbol can precede a floating-point constant. However, it is not part of the constant; it is interpreted as a unary operator.

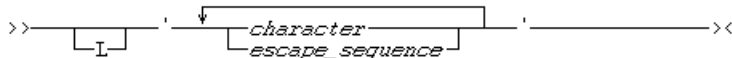
The limits for floating-point values are set in the `/usr/include/float.h` include file. Header files are described in the *AIX Version 4 Files Reference*.

The following are examples of floating-point constants:

Floating-Point Constant	Value
5.3876e4	53876
4e-11	0.00000000004
1e+5	100000
7.321E-3	0.007321
3.2E+4	32000
0.5e-6	0.0000005
0.45	0.45
6.e10	60000000000

## Character Constants

A *character constant* contains a sequence of characters or escape sequences enclosed in single quotation mark symbols.



At least one character or escape sequence must appear in the character constant. The characters can be any from the source program character set, excluding the single quotation mark, backslash and new-line symbols. The prefix **L** indicates a wide character constant. A character constant must appear on a single logical source line.

The value of a character constant containing a single character is the numeric representation of the character in the character set used at run time. The value of a wide character constant containing a single multibyte character is the code for that character, as defined by the **mbtowc** function.

A character constant has type **int**. A wide character constant is represented by a double-byte character of type **wchar\_t**, an integral type defined in the `<stddef.h>` include file. Header files are described in the *AIX Version 4 Files Reference*. Each multibyte character can contain up to 4 bytes.

To represent the single quotation symbol, backslash, and new-line characters, you must use the corresponding escape sequence. For more information on escape sequences, see "Escape Sequences" on page 52.

The following are examples of character constants:

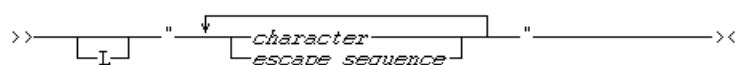
```
'a'      '0'      'x'      '7'      'C'
'\''     '('     '\n'    '\117'
```

## Notes:

1. In **extended** mode, a character constant longer than 2 characters causes a warning to be issued by the C compiler. Only the rightmost 4 characters are used. A character constant with 4 characters has an **unsigned int** value.
2. In **ansi** mode, a character constant longer than 1 character causes a warning to be issued. Only the rightmost 4 characters are used. For example the character constant 'too\_long' causes the following message:  
1506-076 (W) Character constant has more than one character.  
Rightmost four characters are used.

## String Literals

A *string constant* or *literal* contains a sequence of characters or escape sequences enclosed in double quotation mark symbols.



The prefix **L** indicates a wide-character string literal.

A null (`'\0'`) character is appended to each string. For a wide character string (a string prefixed by the letter **L**), the value `'\0'` of type **wchar\_t** is appended. By convention, programs recognize the end of a string by finding the null character.

Multiple spaces contained within a string constant are retained.

To continue a string on the next line, use the line continuation sequence (`\` symbol immediately followed by a new-line character). A carriage return must immediately follow the backslash. In the following example, the string literal `second` causes a compile-time error.

```
char *first = "This string continues onto the next\  
line, where it ends.";          /* compiles successfully. */  
char *second = "The comment makes the \  
invisible to the compiler.";     /* compilation error. */
```

Another way to continue a string is to have two or more consecutive strings. Adjacent string literals are concatenated to produce a single string. You cannot concatenate a wide string constant with a character string constant. For example:

```
"hello " "there"      /* is equivalent to "hello there" */  
"hello " L"there"     /* is not valid */  
"hello" "there"       /* is equivalent to "hellothere" */
```

Characters in concatenated strings remain distinct. For example, the strings `"\xab"` and `"3"` are concatenated to form `"\xab3"`. However, the characters `\xab` and `3` remain distinct and are not merged to form the hexadecimal character `\xab3`.

Following any concatenation, `'\0'` of type **char** is appended at the end of each string. C programs find the end of a string by scanning for this value. For a wide-character string literal, `'\0'` of type **wchar\_t** is appended. For example:

```
char *first = "Hello ";          /* stored as "Hello \0" */  
char *second = "there";         /* stored as "there\0" */  
char *third = "Hello " "there"; /* stored as "Hello there\0" */
```

A character string constant has type *array of char* and static storage duration. A wide character constant has type *array of wchar\_t* and static storage duration.

Use the escape sequence `\n` to represent a new-line character as part of the string. Use the escape sequence `\\` to represent a backslash character as part of the string. You can represent the single quotation mark symbol by itself `'`, but you use the escape sequence `\"` to represent the double quotation mark symbol. For example:

```
#include <stdio.h>
void main ()
{
    char *s = "Hi there! \n";
    char *p = "The backslash character \\\.";
    char *q = "The double quotation mark \".\n";
    printf("%s%s\n%s", s, p, q);
}
```

This program produces the following output:

```
Hi there!
The backslash character \.
The double quotation mark ".
```

You should be careful when modifying string literals because the resulting behavior depends on whether your strings are stored in read/write static memory.

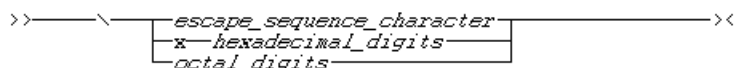
Use the `ro` compiler option or the `#pragma strings` preprocessor directive to change the default storage for string literals. The `#pragma strings` preprocessor directive can also be used to specify whether string literals are readonly or read/write.

The following are examples of string literals:

```
char titles[ ] = "Handel's \"Water Music\"";
char *mail_addr = "Last Name   First Name   MI   Street Address   \
    City   Province   Postal code ";
char *temp_string = "abc" "def" "ghi"; /* *temp_string = "abcdefghi\0" */
wchar_t *wide_string = L"longstring";
```

## Escape Sequences

You can represent any member of the execution character set by an *escape sequence*. They are primarily used to put nonprintable characters in character and string literals. For example, you can use escape sequences to put such characters as tab, carriage return, and backspace into an output stream.



An escape sequence contains a backslash (`\`) symbol followed by one of the escape sequence characters or an octal or hexadecimal number. A hexadecimal escape sequence contains an `x` followed by one or more hexadecimal digits (0-9, A-F, a-f). An octal escape sequence uses up to three octal digits (0-7). The value of the hexadecimal or octal number specifies the value of the desired character or wide character.

**Note:** The line continuation sequence (`\` followed by a new-line character) is not an escape sequence. It is used in character strings to indicate that the current line continues on the next line.

The escape sequences and the characters they represent are:

Escape Sequence	Character Represented
<code>\a</code>	Alert (bell, alarm)
<code>\b</code>	Backspace
<code>\f</code>	Form feed (new page)
<code>\n</code>	New-line



<code>\r</code>	Carriage return
<code>\t</code>	Horizontal tab
<code>\v</code>	Vertical tab
<code>\'</code>	Single quotation mark
<code>\"</code>	Double quotation mark
<code>\?</code>	Question mark
<code>\\</code>	Backslash

The value of an escape sequence represents the member of the character set used at run time. Escape sequences are translated during preprocessing. For example, the AIX Version 4 operating system uses the ASCII character set, where the value of the escape sequence `\x56` is the letter V.

Use escape sequences only in character constants or in string literals.

If an escape sequence is not recognized, the compiler removes the backslash and issues a warning message. For example, the string `"abc\def"` becomes `"abcdef"`. Note that this behavior is implementation-defined.

When a hexadecimal escape sequence is longer than two digits, the compiler issues a warning. Only the rightmost two digits are used. For example, in the following statement

```
printf ("\x06asset \n");
```

only the digits 6a are retained.

In string and character sequences, when you want the backslash to represent itself (rather than the beginning of an escape sequence), you must use a `\\` backslash escape sequence.

```
#include <stdio.h>
void main()
{
    char a,b,c,d,e;
    a='a';
    b=97;      /* ASCII integer value */
    c='\141';  /* ASCII octal value */
    d='\x61';  /* ASCII hexadecimal value */
    e='\n';
    printf("%c %c %c %c %c\n", a, b, c, d, e);
}
```

### Related Reading

- “Constant Expressions” on page 71
- “Tokens” on page 45
- “Comments” on page 46
- “Identifiers” on page 47
- “Type Specifiers” on page 66
- “#pragma strings Preprocessor Directive” on page 376
- “ro” on page 317 Compiler Option

---

## Identifier Behavior in Your Program

### Scope of Identifier Visibility

An identifier becomes *visible* with its declaration. The region where an identifier is visible is referred to as the identifier’s *scope*.

The scope of an identifier is determined by where the identifier is declared. The four kinds of scope and their descriptions are:

### Block Scope

The identifier's declaration is located inside a statement block.

A block starts with an opening brace (`{`) and ends with a closing brace (`}`). An identifier with block scope is visible between the point where it is declared and the closing brace that ends the block.

### Function Scope

Block scope is sometimes referred to as *local* scope.  
The only identifier with function scope is a label name.

A label is implicitly declared by its appearance in the program source. A **goto** statement transfers control to the label specified in the **goto** statement. The label is visible to any **goto** statement that appears in the same function as the label.

### File Scope

The identifier's declaration appears outside any block.

It is visible from the point where it is declared to the end of the source file. If the source files are included by **#include** preprocessor directives, those files are considered to be part of the source, and the identifier will be visible to all included files that appear after the declaration of the identifier.

The identifier can be declared again as a block scope variable. The new declaration replaces the file-scope declaration until the end of the block.

### Function Prototype Scope

The identifier's declaration appears within the list of parameters in a function prototype.

It is visible from the point where it is declared to the closing parenthesis of the prototype declaration.

### Related Reading

"Program Linkage Between Identifiers"

"Storage Duration" on page 57

"Name Spaces" on page 57

"Identifiers" on page 47

"#include Preprocessor Directive" on page 356

"goto" on page 151 Statement

## Program Linkage Between Identifiers

The association, or lack of association, between two identical identifiers is known as *linkage*. The kind of linkage that an identifier has depends on the way that it is declared.

Types of identifier linkage and their descriptions are:

### Internal Linkage

Internal linkage occurs where identical identifiers within a single source file refer to the same data object or function.

The following kinds of labels have internal linkage:

- All identifiers with file or block scope that have the keyword **static** in their declarations. Functions with **static** storage class are visible only in the source file in which you define them.
- C identifiers declared at file scope with the specifier **const**, and not explicitly declared **extern**.

A variable that has **static** storage class can be defined within a block or outside of a function.

- If the definition occurs within the block, the variable has internal linkage and is visible only within the block after its declaration is seen.
- If the definition occurs outside of a function, the variable has internal linkage, and is available from the point where it is defined until the end of the current source file.

A class name that has no static members or non-inline member functions, and that has not been used in the declaration of an object, function, or class, is local to its translation unit.

If the declaration of an identifier has the keyword **extern**, and if a previous declaration of the identifier is visible at file scope, the identifier has the same linkage as the first declaration.

### External Linkage

External linkage occurs where identical identifiers in separately compiled files refer to the same data object or function.

The following kinds of identifiers have external linkage:

- Identifiers with file or block scope that have the keyword **extern** in their declarations.

If a previous declaration of the identifier is visible at file scope, the identifier has the same linkage as the first declaration. For example, a variable or function that is first declared with the keyword **static** and is later declared with the keyword **extern** has internal linkage.

- Function identifiers declared without storage-class specifiers.
- Object identifiers that have file scope declared without a storage-class specified. Storage is allocated for such object identifiers.
- Static class members and non-inline member functions.

Identifiers declared with the keyword **extern** can be defined in other translation units.

### No Linkage

No linkage occurs where each identical identifier refers to a unique object.

The following kinds of identifiers have no linkage:

- Identifiers that do not represent an object or a function, including labels, enumerators, **typedef** names, type names, and template names.
- Identifiers that represent a function argument.
- Identifiers declared inside a block without the keyword **extern**.

### Related Reading

“Scope of Identifier Visibility” on page 53

“Example of File and Function Prototype Visibility Scopes” on page 56

“Example of File and Block Visibility Scopes” on page 56

“Data Type Qualifiers” on page 115

“extern” on page 109

“static” on page 112

“typedef” on page 115

## Example of File and Function Prototype Visibility Scopes

In the following example, the variable `x`, which is declared on line 1, is different from the `x` declared on line 2. The variable declared on line 2 has function prototype scope and is visible only up to the closing parenthesis of the prototype declaration. Visibility of the variable `x` declared on line 1 resumes after the end of the prototype declaration.

```
1 int x = 4;           /* variable x defined with file scope */
2 long myfunc(int x, long y); /* variable x has function */
3                       /* prototype scope */
4 int main(void)
5 {
6     /* . . . */
7 }
```

### Related Reading

“Scope of Identifier Visibility” on page 53

“Example of File and Block Visibility Scopes”

## Example of File and Block Visibility Scopes

Functions with `static` storage class are visible only in the source file they are defined in. All other functions can be globally visible.

The following program illustrates blocks, nesting, and scope. The example shows two kinds of scope: file and block. The `main` function prints the values 1, 2, 3, 0, 3, 2, 1 on separate lines. Each instance of `i` represents a different variable.

```
#include <stdio.h>
int i = 1;           /* i defined at file scope */
int main(int argc, char * argv[])
+--- {
    printf("%d\n", i);           /* Prints 1 */
    +--- {
        int i = 2, j = 3;       /* i and j defined at
                                block scope */
        printf("%d\n%d\n", i, j); /* Prints 2, 3 */
        +--- {
            int i = 0;          /* i is redefined in a nested block */
                                /* previous definitions of i are hidden */
            printf("%d\n%d\n", i, j); /* Prints 0, 3 */
        +--- }
        printf("%d\n", i);       /* Prints 2 */
    +--- }
    printf("%d\n", i);           /* Prints 1 */
    return 0;
+--- }
```

### RELATED CONCEPTS

“Scope of Identifier Visibility” on page 53

## RELATED TASKS

“Example of File and Function Prototype Visibility Scopes” on page 56

## Related Reading

“Scope of Identifier Visibility” on page 53

“Example of File and Function Prototype Visibility Scopes” on page 56

## Storage Duration

*Storage duration* determines how long storage for an object exists. An object has either *static* storage duration or *automatic* storage duration depending on its declaration. Descriptions of each follow:

**Static storage** Is allocated at initialization and remains available until the program ends. Objects have static storage duration if they:

- Have file scope
- Have external or internal linkage OR
- Contain the **static** storage class specifier.

**Automatic storage** Is allocated and removed according to the scope of the identifier. Objects have automatic storage duration if they are:

- Parameters in a function definition.
- Declared at block scope and do not have any storage class specifier, or,
- Declared at block scope and have the **register** or **auto** storage class specifier.

For example, storage for an object declared at block scope is allocated when the identifier is declared and removed when the closing brace (}) is reached.

**Note:** Objects can also have *heap* storage duration. Heap objects are created at runtime and storage is allocated for them by calling a function such as **malloc()**.

## Related Reading

“Scope of Identifier Visibility” on page 53

“Program Linkage Between Identifiers” on page 54

“Name Spaces”

“auto” on page 106 Storage Class Specifier

“register” on page 111 Storage Class Specifier

“static” on page 112 Storage Class Specifier

## Name Spaces

The compiler sets up *name spaces* to distinguish among identifiers referring to different kinds of entities. Identical identifiers in different name spaces do not interfere with each other, even if they are in the same scope.

You must assign unique names within each name space to avoid conflict. The same identifier can be used to declare different objects as long as each identifier is unique within its name space. The syntactic context of an identifier within a program lets the compiler resolve its name space without ambiguity.

Identifiers in the same name space can be redefined within enclosed program blocks, as described in “Scope of Identifier Visibility” on page 53.

Within each of the following four name spaces, the identifiers must be unique.

- *Tags* of these types must be unique within a single scope:
  - Enumerations
  - Structures and unions

- *Members* of structures and unions must be unique within a single structure or union type.
- *Statement labels* have function scope and must be unique within a function.
- All other *ordinary identifiers* must be unique within a single scope:
  - Function names
  - Variable names
  - Names of function parameters
  - Enumeration constants
  - **typedef** names.

#### Related Reading

“Scope of Identifier Visibility” on page 53

“Example of Name Space Separation”

### Example of Name Space Separation

Structure tags, structure members, variable names, and statement labels are in four different name spaces; no conflict occurs among the four items named `student` in the following example:

```
int get_item()
{
    struct student      /* structure tag      */
    {
        char student[20]; /* structure member */
        int section;
        int id;
    } student;          /* structure variable */
    goto student;
    student: ;          /* null statement label */
    return (0);
}
```

Each occurrence of `student` is interpreted by its context in the program. For example, when `student` appears after the keyword **struct**, it is a structure tag. When `student` appears after either of the member selection operators `.` or `->`, the name refers to the structure member. When `student` appears after the **goto** statement, control is passed to the null statement label. In other contexts, the identifier `student` refers to the structure variable.

#### Related Reading

“Name Spaces” on page 57

“Scope of Identifier Visibility” on page 53

---

## Preprocessor Directives

*Preprocessing* is a step that takes place before compilation that lets you:

- Replace tokens in the current file with specified replacement tokens.
- Imbed files within the current file
- Conditionally compile sections of the current file
- Generate diagnostic messages
- Change the line number of the next line of source and change the file name of the current file.

A *token* is a series of characters delimited by white space. The only white space allowed on a preprocessor directive is the space, horizontal tab, vertical tab, form feed, and comments. The new-line character can also separate preprocessor tokens.

The preprocessed source program file must be a valid C program.

Preprocessor directives begin with the # token followed by a preprocessor keyword. The # token must appear as the first character that is not white space on a line. The # is not part of the directive name and can be separated from the name with white spaces.

A preprocessor directive ends at the new-line character unless the last character of the line is the \ (backslash) character. If the \ character appears as the last character in the preprocessor line, the preprocessor interprets the \ and the new-line character as a continuation marker. The preprocessor deletes the \ (and the following new-line character) and splices the physical source lines into continuous logical lines.

Except for some **#pragma** directives, preprocessor directives can appear anywhere in a program.

#### Related Reading

“Preprocessing Operations”

“Preprocessor Macros”

“Conditional Compilation Directives” on page 60

“List of Standard Preprocessor Directives” on page 347

“#pragma Preprocessor Directives” on page 363

## Preprocessing Operations

Preprocessing carries out the following operations on your program source files:

1. New-line characters are introduced as needed to replace system-dependent end-of-line characters, and other system-dependent character-set translations are performed as needed. Trigraph sequences are replaced by equivalent single characters.
2. Each \ (backslash) followed by a new-line character is deleted, and the next source line is appended to the line that contained the backslash.
3. The source text is decomposed into preprocessing tokens and sequences of white space. A single white space replaces each comment. A source file cannot end with a partial token or comment.
4. Preprocessing directives are run, and macros are expanded.
5. Escape sequences in character constants and string literals are replaced by their equivalent values.
6. Adjacent string literals are concatenated.

The rest of the compilation process operates on the preprocessor output, which is syntactically and semantically analyzed and translated, and then linked as necessary with other programs and libraries.

#### Related Reading

“Preprocessor Directives” on page 58

“Preprocessor Macros”

“Conditional Compilation Directives” on page 60

“C Programming Character Set” on page 160

## Preprocessor Macros

You can use the **#define** preprocessor directive to define a macro that assigns a value to an identifier. The preprocessor replaces subsequent occurrences of that identifier with its assigned value until the identifier is undefined with the **#undef** preprocessor directive, or until the end of the program source is reached, whichever comes first.

There are two basic types of macro definitions that you can use to assign a value to an identifier:

#### Object-like Macros

Replaces a single identifier with a specified token, or constant value.

## Function-like Macros

Associates a user-defined function and argument list to an identifier. When the preprocessor encounters that identifier in the program source, the defined function is inserted in place of the identifier along with any corresponding arguments.

### Related Reading

“Preprocessor Directives” on page 58

“Preprocessing Operations” on page 59

“Conditional Compilation Directives”

“#define Preprocessor Directive” on page 348

“#undef Preprocessor Directive” on page 358

“Predefined Preprocessor Macros” on page 359

“Preprocessor Macro Operators” on page 377

## Conditional Compilation Directives

A *preprocessor conditional compilation directive* causes the preprocessor to conditionally suppress the compilation of portions of source code. These directives test a constant expression or an identifier to determine which tokens the preprocessor should pass on to the compiler and which tokens should be bypassed during preprocessing. The directives are:

- **#if**
- **#ifdef**
- **#ifndef**
- **#else**
- **#elif**
- **#endif**

For each **#if**, **#ifdef**, and **#ifndef** directive, there are zero or more **#elif** directives, zero or one **#else** directive, and one matching **#endif** directive. All the matching directives are considered to be at the same nesting level.

You can nest conditional compilation directives. In the following directives, the first **#else** is matched with the **#if** directive.

```
#ifdef MACNAME
    /* tokens added if MACNAME is defined          */
#   if TEST <=10
    /* tokens added if MACNAME is defined and TEST <="10" */
#   else
    /* tokens added if MACNAME is defined and TEST> 10  */
#   endif
#else
    /* tokens added if MACNAME is not defined        */
#endif
```

Each directive controls the block immediately following it. A block consists of all the tokens starting on the line following the directive and ending at the next conditional compilation directive at the same nesting level.

Each directive is processed in the order in which it is encountered. If an expression evaluates to zero, the block following the directive is ignored.

When a block following a preprocessor directive is to be ignored, the tokens are examined only to identify preprocessor directives within that block so that the conditional nesting level can be determined. All tokens other than the name of the directive are ignored.

Only the first block whose expression is nonzero is processed. The remaining blocks at that nesting level are ignored. If none of the blocks at that nesting level has been processed and there is a **#else** directive,



the block following the **#else** directive is processed. If none of the blocks at that nesting level has been processed and there is no **#else** directive, the entire nesting level is ignored.

### Related Reading

- “Preprocessor Directives” on page 58
- “Preprocessing Operations” on page 59
- “Preprocessor Macros” on page 59
- “Examples of Conditional Preprocessor Directives” on page 355
- “#if, #elif Preprocessor Directives” on page 352
- “#else Preprocessor Directive” on page 353
- “#endif Preprocessor Directive” on page 353
- “#if, #elif Preprocessor Directives” on page 352
- “#ifdef Preprocessor Directive” on page 354
- “#ifndef Preprocessor Directive” on page 355
- “List of Standard Preprocessor Directives” on page 347

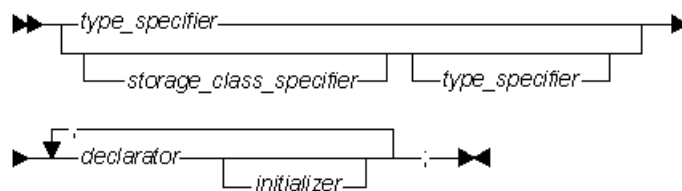
---

## Declarations Overview

A *declaration* establishes the names and characteristics of data objects and functions used in a program. A *definition* allocates storage for data objects or specifies the body for a function. When you define a type, no storage is allocated.

Declarations determine the following properties of data objects and their identifiers:

- Scope, which describes the visibility of an identifier in a block or source file.
- Linkage, which describes the association between two identical identifiers.
- Storage duration, which describes when the system allocates and frees storage for a data object.
- Type, which describes the kind of data the object is to represent.



The declaration for a data object can include the following components:

- Qualifier and declarator
- Storage class
- Initializer
- Type specifier

The following table shows examples of declarations and definitions. The identifiers declared in the first column do not allocate storage; they refer to a corresponding definition. In the case of a function, the corresponding definition is the code or body of the function. The identifiers declared in the second column allocate storage; they are both declarations and definitions.

### Declarations

```
extern double pi;  
float square(float x);
```

### Declarations and Definitions

```
double pi = 3.14159265;  
float square(float x) { return x*x; }
```

## Declarations

```
struct payroll;
```

## Declarations and Definitions

```
struct payroll {  
    char *name;  
    float salary;  
} employee;
```

## Related Reading

“Program Linkage Between Identifiers” on page 54

“Scope of Identifier Visibility” on page 53

“Storage Duration” on page 57

“Block Scope Data Declarations”

“File Scope Data Declarations” on page 63

“Declarators” on page 64

“Storage Class Specifiers” on page 65

“Initializers” on page 65

“Type Specifiers” on page 66

---

## Block Scope Data Declarations

In C, a *block scope data declaration* can only be put at the beginning of a block. It describes a variable and makes that variable accessible to the current block. All block scope declarations that do not have the **extern** storage class specifier are definitions and allocate storage for that object.

You can declare a data object with block scope with any one of the following storage class specifiers:

- **auto**
- **extern**
- **register**
- **static**
- **typedef**

If you do not specify a storage class specifier in a block-scope data declaration, the default storage class specifier **auto** is used. If you specify a storage class specifier, you can omit the type specifier. If you omit the type specifier, all variables in that declaration receive type **int**.

## Initialization

You cannot initialize a variable declared in a block scope data declaration that has the **extern** storage class specifier.

The types of variables you can initialize and the values that uninitialized variables receive vary for that storage class specifier. See “Storage Class Specifiers” on page 65 for details on the different storage classes.

## Storage

The duration and type of storage varies for each storage class specifier.

Declarations with the **auto** or **register** storage class specifier result in automatic storage duration. Declarations with the **extern** or **static** storage class specifier result in static storage duration.

#### Related Reading

“Declarations Overview” on page 61  
“File Scope Data Declarations”  
“Declarators” on page 64  
“Storage Class Specifiers” on page 65  
“Initializers” on page 65  
“Type Specifiers” on page 66  
“auto” on page 106  
“extern” on page 109  
“register” on page 111  
“static” on page 112  
“int, long, short” on page 81  
“typedef” on page 115

## File Scope Data Declarations

A *file scope data declaration* appears outside any function definition. It describes a variable and makes that variable accessible to all functions that are in the same file and whose definitions appear after the declaration.

A *file scope data definition* is a data declaration at file scope that also causes storage to be allocated for that variable. All objects whose identifiers are declared at file scope have static storage duration.

Use a file scope data declaration to declare variables that you want to have external linkage.

The only storage class specifiers you can put in a file scope data declaration are **static**, **extern**, and **typedef**. If you specify **static**, all variables defined in it have internal linkage. If you do not specify **static**, all variables defined in it have external linkage. If you specify the storage class you can omit the type specifier. If you omit the type specifier, all variables defined in that declaration receive the type **int**.

### Initialization

You can initialize any object with file scope. If you do not initialize a file scope variable, its initial value is zero of the appropriate type. If you do initialize it, the initializer must be described by a constant expression, or it must reduce to the address of a previously declared variable at file scope, possibly modified by a constant expression. Initialization of all variables at file scope takes place before the main function begins running.

### Storage

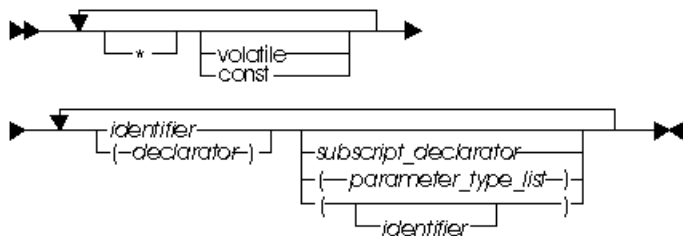
All objects with file scope data declarations have static storage duration. Storage is allocated at runtime and freed when the program stops running.

#### Related Reading

“Declarations Overview” on page 61  
“Block Scope Data Declarations” on page 62  
“Declarators” on page 64  
“Storage Class Specifiers” on page 65  
“Initializers” on page 65  
“Type Specifiers” on page 66  
“int, long, short” on page 81  
“extern” on page 109  
“static” on page 112  
“typedef” on page 115

## Declarators

A *declarator* designates a data object or function. Declarators appear in all data definitions and declarations, and in some type definitions.



You cannot declare or define a **volatile** or **const** function.

A *subscript declarator* describes the number of dimensions in an array, and the number of elements in each dimension.

A simple declarator consists of an identifier, which names a data object. For example, the following block scope data declaration uses `initial` as the declarator:

```
auto char initial
```

The data object `initial` has the storage class **auto**, and the data type **char**.

The following table describes some more declarators:

### Example

```
int owner
int *node
int names[126]
int *action( )
volatile int min
int * volatile volume
volatile int * next
volatile int * sequence[5]
extern const volatile int op_system_clock
```

### Description

`owner` is an **int** object.  
`node` is a pointer to an **int** data object.  
`names` is an array of 126 **int** elements.  
`action` is a function returning a pointer to an **int**.  
`min` is an **int** that has the volatile qualifier.  
`volume` is a **volatile** pointer to an **int**.  
`next` is a pointer to a **volatile int**.  
`sequence` is an array of five pointers to **volatile int** objects.  
`op_system_clock` is a constant and **volatile int** with static storage duration and external linkage.

### Related Reading

“Declarations Overview” on page 61  
“Block Scope Data Declarations” on page 62  
“File Scope Data Declarations” on page 63  
“Storage Class Specifiers” on page 65  
“Initializers” on page 65  
“Type Specifiers” on page 66  
“Arrays” on page 86  
“int, long, short” on page 81  
“char” on page 79  
“auto” on page 106  
“Data Type Qualifiers” on page 115  
“Data Type Qualifiers” on page 115

## Storage Class Specifiers

The storage class specifier used within the declaration determines whether:

- The object has internal, external, or no linkage.
- The object is to be stored in memory or in a register, if available.
- The object receives the default initial value 0 or an indeterminate default initial value.
- The object can be referenced throughout a program or only within the function, block, or source file where the variable is defined.
- The storage duration for the object is static (storage is maintained throughout program run time) or automatic (storage is maintained only during the execution of the block where the object is defined).

For a function, the storage class specifier determines the linkage of the function.

Declarations with the **auto** or **register** storage-class specifier result in automatic storage. Those with the **extern** or **static** storage-class specifier result in static storage.

Most local declarations that do not include the **extern** storage-class specifier allocate storage; however, function declarations and type declarations do not allocate storage.

The only storage-class specifiers allowed in a global or file scope declaration are **static** and **extern**.

Storage class specifier keywords are:

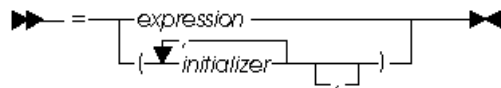
- **auto**
- **extern**
- **register**
- **static**
- **typedef**

### Related Reading

“auto” on page 106  
“extern” on page 109  
“register” on page 111  
“static” on page 112  
“typedef” on page 115

## Initializers

An *initializer* is an optional part of a data declaration that specifies an initial value of a data object.



The initializer consists of the = symbol followed by an initial *expression* or a braced list of initial expressions separated by commas. The number of initializers must not be more than the number of elements to be initialized. An initializer list with fewer initializers than elements, can end with a comma, indicating that the rest of the uninitialized elements are initialized to zero. The initial expression evaluates to the first value of the data object.

To assign a value to a scalar object, use the simple initializer: = *expression*. For example, the following data definition uses the initializer = 3 to set the initial value of group to 3:

```
int group = 3;
```

For unions and structures, the set of initial expressions must be enclosed in { } (braces) unless the initializer is a string literal. If the initializer of a character string is a string literal, the { } are optional. Individual expressions must be separated by commas, and groups of expressions can be enclosed in braces and separated by commas.

In an array, structure, or union initialized using a brace-enclosed initializer list, any members or subscripts that are not initialized are implicitly initialized to zero of the appropriate type.

The initialization properties of each data type are described in the section for that data type.

In the following example, only the first eight elements of the array `grid` are explicitly initialized. The remaining four elements that are not explicitly initialized are initialized as if they were explicitly initialized to zero.

```
static short grid[3][4] = {0, 0, 0, 1, 0, 0, 1, 1};
```

The following example is an equivalent initialization of the array `grid`:

```
static short grid[3][4] = {{0, 0, 0, 1},
                          {0, 0, 1, 1}};
```

The initial values of `grid` are:

Element	Value	Element	Value
<code>grid[0][0]</code>	0	<code>grid[1][2]</code>	1
<code>grid[0][1]</code>	0	<code>grid[1][3]</code>	1
<code>grid[0][2]</code>	0	<code>grid[2][0]</code>	0
<code>grid[0][3]</code>	1	<code>grid[2][1]</code>	0
<code>grid[1][0]</code>	0	<code>grid[2][2]</code>	0
<code>grid[1][1]</code>	0	<code>grid[2][3]</code>	0

### Related Reading

- “Declarations Overview” on page 61
- “Block Scope Data Declarations” on page 62
- “File Scope Data Declarations” on page 63
- “Declarators” on page 64
- “Storage Class Specifiers” on page 65
- “Type Specifiers”

## Type Specifiers

Type specifiers indicate the type of object or function being created.

The basic range of types are:

- “char” on page 79
- “float, double” on page 80
- “int, long, short” on page 81
- “enum” on page 82
- “void” on page 85

You can use the basic types listed above to derive the following additional object types:

- “Pointers” on page 90
- “Arrays” on page 86
- “struct (Structures)” on page 95

- “union (Unions)” on page 103
- “Functions” on page 77

The integral types are `char` and `int` of all sizes. Floating-point numbers can have types **float**, **double**, or **long double**. Integral and floating-point types are collectively called *arithmetic* types.

You can give names to both basic and derived types with the **typedef** specifier.

#### Related Reading

- “Declarations Overview” on page 61
- “Block Scope Data Declarations” on page 62
- “File Scope Data Declarations” on page 63
- “Declarators” on page 64
- “Storage Class Specifiers” on page 65
- “Initializers” on page 65
- “Type Specifiers” on page 66
- “Character Constants” on page 50
- “Integer Constant” on page 48
- “Floating-Point Constants” on page 49
- “typedef” on page 115

---

## Expressions and Operators

Expressions are sequences of operators, operands, and punctuators that specify a computation. The evaluation of an expressions is based on the operators that the expression contains, and the context in which the operators are used.

#### Related Reading

- “Operator Precedence and Associativity”
- “Operands” on page 69
- “lvalues” on page 70
- “Types of Expressions” on page 70
- “Constant Expressions” on page 71
- “Function Calls” on page 72
- “Operator Precedence and Associativity Table” on page 117
- “Primary Operators” on page 118
- “Unary Operators” on page 120
- “Binary Operators” on page 124
- “Assignment Operators” on page 133
- “Comma Operator ( , )” on page 136

## Operator Precedence and Associativity

Two characteristics of operators determine how they will group with operands:

<i>precedence</i>	Precedence is the priority for grouping different types of operators with their operands.
<i>associativity</i>	Associativity is the left-to-right or right-to-left order for grouping operands to operators that have the same precedence.

For example, in the following statements, the value of 5 is assigned to both `a` and `b` because of the right-to-left associativity of the `=` operator. The value of `c` is assigned to `b` first, and then the value of `b` is assigned to `a`.

```
b = 9;
c = 5;
a = b = c;
```

Because the order of the expression evaluation is not specified, you can explicitly force the grouping of operands with operators by using parentheses. In the expression:

```
a + b * c / d
```

the `*` and `/` operations are performed before the `+` because of precedence. Further, `b` is multiplied by `c` before it is divided by `d` because of associativity.

## Special Cases

Order of evaluation for function call arguments or for the operands of binary operators is not specified. Avoid writing ambiguous expressions, such as:

```
z = (x * ++y) / func1(y);
func2(++i, x[i]);
```

In the example above, the order of evaluation of `++y` and `func1(y)` is not defined. In fact, they might not even be evaluated in the same order at different optimization levels. Do not write code that depends on a particular order of evaluation of operators that have the same precedence.

The order of grouping operands with operators in an expression containing more than one instance of an operator with both associative and commutative properties is not specified. The operators that have the same associative and commutative properties are `*`, `+`, `&`, `|`, and `^`.

The order of evaluation for the operands of the logical AND (`&&`) and OR (`||`) operators is always left-to-right. If the operand on the left side of a `&&` operator evaluates to a 0 (zero), the operand on the right side is not evaluated. If the operand on the left side of a `||` operator evaluates to a non-zero value, the operand on the right side is not evaluated.

### Related Reading

“Examples of Operator Precedence and Associativity”

“Operator Precedence and Associativity Table” on page 117

## Examples of Operator Precedence and Associativity

The parentheses in the following expressions explicitly show how the compiler groups operands and operators. If the parentheses did not appear in these expressions, the operands and operators are grouped in the same manner as indicated by the parentheses.

```
total = (4 + (5 * 3));
total = (((8 * 5) / 10) / 3);
total = (10 + (5 / 3));
```

Because the order of grouping operands with operators that are both associative and commutative is not specified, the compiler can group the operands and operators in the expression:

```
total = price + prov_tax + city_tax;
```

in the following ways, as indicated by the parentheses:



```
total = price + prov_tax + city_tax;
total = price + prov_tax + city_tax;
total = price + prov_tax + city_tax;
```

If the values in this expression are integers, the grouping of operands and operators does not affect the result. Different groupings of floating-point operators, however, may give different results because intermediate values are rounded.

In certain expressions, the grouping of operands and operators can affect the result. For example, in the following expression, each of the three function calls might modify the same global variables.

```
a = b() + c() + d();
```

This expression might give different results, depending on the order in which the functions are called.

If the expression contains operators that are both associative and commutative, and the order of grouping operands with operators can affect the result of the expression, separate the expression into several expressions. For example, the following expressions could replace the previous expression if the called functions do not produce any side effects that affect the variable `a`.

```
a = b();
a += c();
a += d();
```

Integer overflows are ignored. Division by zero and floating-point exceptions are implementation-dependent.

### Related Reading

“Operator Precedence and Associativity” on page 67

“Operator Precedence and Associativity Table” on page 117

## Operands

Most expressions can contain several different, but related, types of operands. The following *type classes* described related types of operands.

<i>integral</i>	Character objects and constants, objects having an enumeration type, and objects having the types <b>short</b> , <b>int</b> , <b>long</b> , <b>long long</b> , <b>unsigned short</b> , <b>unsigned int</b> , <b>unsigned long</b> , or <b>unsigned long long</b> .
<i>arithmetic</i>	Integral objects listed above, and objects having the types <b>float</b> , <b>double</b> , <b>long double</b> , and <b>long float</b> .
<i>scalar</i>	Arithmetic objects listed above, and pointers to any object type.
<i>aggregate</i>	Arrays, structures, and unions.

Many operators cause conversions from one data type to another.

### Related Reading

“Integral Promotions” on page 74

“Standard Type Conversions” on page 74

“Arithmetic Conversions” on page 76

“Arithmetic Conversions Table” on page 137

“Arithmetic Conversions for extended Level C” on page 167

“int, long, short” on page 81

“float, double” on page 80

## Ivalues

An *lvalue* is an expression that represents an object. A *modifiable lvalue* is an expression representing an object that can be changed. It is typically the left operand in an assignment expression. For example, arrays and **const** objects are not modifiable lvalues, but **static int** objects are.

All assignment operators evaluate their right operand and assign that value to the left operand. The left operand must evaluate to a reference to an object.

The address operator (&) requires an lvalue as an operand, while the increment (++) and the decrement (—) operators require a modifiable lvalue as an operand.

Expression	Lvalue of Expression
x = 42;	x
*ptr = newvalue;	*ptr
a++	a

### Related Reading

“Expressions and Operators” on page 67

“Operator Precedence and Associativity” on page 67

“Operands” on page 69

“Types of Expressions”

“Data Type Qualifiers” on page 115

“int, long, short” on page 81

“static” on page 112

## Types of Expressions

**Primary Expressions** A *primary expression* can be any of the following:

- identifier
- string literal
- parenthesized expression
- constant expression
- function call
- array element specification
- structure of union member specification

All primary operators have the same precedence, and have left-to-right associativity.

**Unary Expressions** A *unary expression* contains one operand and a unary operator.

All unary operators have the same precedence, and have right-to-left associativity. The usual arithmetic conversions are performed on the operands of most unary expressions.

**Binary Expressions** A *binary expression* contains two operands separated by one operator.

Not all binary operators have the same precedence. All binary operators have left-to-right associativity. The usual arithmetic conversions are performed on the operands of most binary expressions.

The order in which the operands of most binary operators are evaluated is not specified. To ensure correct results, avoid creating binary expressions that depend on the order in which the compiler evaluates the operands.

## Conditional Expressions

A *conditional expressions* is a compound expression that contains a condition (*operand1*), an expression to be evaluated if the condition has a non-zero value (*operand2*), and an expression to be evaluated if the condition has the value 0 (*operand3*).

Conditional expressions have right-to-left associativity. The left operand (*operand1*) is evaluated first, and then only one of the two remaining operands is evaluated. If that operand's expression contains or returns arithmetic types, the usual arithmetic conversions are performed on that expression's values.

## Assignment Expressions

An *assignment expression* stores a value in the object designated by the left operand. There are two types of assignment operators: simple assignment, and compound assignment.

The left operand in all assignment expressions must be a modifiable lvalue. The type of the expression is the type of the left operand. The value of the expression is the value of the left operand after the assignment completes. The result of an assignment expression is not an lvalue.

## Comma Expression

All assignment operators have the same precedence, and have right-to-left associativity.

A *comma expression* contains two operands separated by a comma. Although the compiler evaluates both operands, the value of the right operand is the value of the expression. The left operand is evaluated, possibly producing side effects, and the value is discarded. The result of a comma expression is not an lvalue.

Both operands of a comma expression can have any type. All comma expressions have left-to-right associativity. The left operand is fully evaluated before the right operand.

## Related Reading

"Operator Precedence and Associativity" on page 67

"Operands" on page 69

"lvalues" on page 70

"Constant Expressions"

"Function Calls" on page 72

"Expressions" on page 149

"Operator Precedence and Associativity Table" on page 117

"Primary Operators" on page 118

"Unary Operators" on page 120

"Binary Operators" on page 124

"Conditional Operator (?)" on page 131

"Assignment Operators" on page 133

"Comma Operator ( , )" on page 136

## Constant Expressions

A *constant expression* is an expression with a value that is determined during compilation. That value can be evaluated at runtime, but cannot be changed. Constant expressions can be composed of integer, character, floating-point, and enumeration constants, as well as other constant expressions. Some constant expressions, such as string literals or address constants, are lvalues.

The C language requires constants in the following places:

- In the subscript declarator, as the description of the array bound
- After the keyword **case** in a **switch** statement
- In an enumerator, as the numeric value of an **enum** constant
- In a bit-field width specifier
- In the preprocessor **#if** statement (enumeration constants, address constants, and **sizeof** cannot be used in the preprocessor **#if** statement)
- In the initializer of a file scope data definition

In all of these contexts, except for an initializer of a file scope data definition, the constant expression can contain integer, character, and enumeration constants, casts to integral types, and **sizeof** expressions. Function-scope **static** and **extern** declarations can be initialized with the address of a previously-defined **static** or **extern**.

In a file scope data definition, the initializer must evaluate to a constant or to the address of a static storage (**extern** or **static**) object (plus or minus an integer constant) that is defined or declared earlier in the file. The constant expression in the initializer can contain integer, character, enumeration, or float constants, casts to any type, **sizeof** expressions, and unary address expressions.

The following show constants used in expressions:

Expression Containing Constant	Constant
<code>x = 42;</code>	42
<code>extern int cost = 1000;</code>	1000
<code>y = 3 * 29;</code>	3 * 29

### Related Reading

“Types of Expressions” on page 70

“switch” on page 155

“enum” on page 82

“#if, #elif Preprocessor Directives” on page 352

sizeof (page 122)

“extern” on page 109

“static” on page 112

## Function Calls

A *function call* is a primary expression containing a simple type name and a parenthesized argument list. The argument list can contain any number of expressions separated by commas. It can also be empty. For example:

```
stub()  
overdue(account, date, amount)  
notify(name, date + 5)  
report(error, time, date, ++num)
```

The arguments are evaluated, and each formal parameter is assigned the value of the corresponding argument. Assigning a value to a formal parameter within the function body changes the value of the parameter within the function, but has no effect on the argument.

The type of a function call expression is the return type of the function. The return value is determined by the return statement in the function definition. The result of a function call is an lvalue only if the function returns a reference. A function can call itself.

If you want a function to change the value of a variable, pass a pointer to the variable you want changed. When a pointer is passed as a parameter, the pointer and not the object pointed to is copied.

Arguments that are arrays and functions are converted to pointers before being passed as function arguments.

Arguments passed to non-prototyped C functions undergo conversions. Type **short** or **char** parameters are converted to **int**, and **float** parameters are converted to **double**. Use a cast expression for other conversions.

The compiler compares the data types provided by the calling function with the data types that the called function expects. The compiler might also perform type conversions if the declaration of the function is:

- in function prototype format and the parameters differ from the prototype, or,
- visible at the point where the function is called.

### Related Reading

“Functions” on page 77

“Types of Expressions” on page 70

“Operator Precedence and Associativity” on page 67

“Operands” on page 69

“lvalues” on page 70

“Functions” on page 77

“Examples of Function Calls”

“Example of the main() Function” on page 145

“Examples of Function Declarations” on page 139

“Examples of Function Definitions” on page 142

“main() Function” on page 144

“Function Declarations” on page 138

“Function Definitions” on page 139

“return” on page 154

“char” on page 79

“int, long, short” on page 81

“float, double” on page 80

## Examples of Function Calls

For example, the declaration of `funct` is a prototype. When function `funct` is called, the parameter `f` is converted to a **double**, and parameter `c` is converted to an **int**.

```
char * funct (double d, int i);
main
{
    float f;
    char c;
    funct(f, c) /* f is a double, c is an int */
}
```

The order in which parameters are evaluated is not specified. Avoid calls such as:

```
method(sample1, batch.process-, batch.process);
```

In this example, `batch.process-` might be evaluated last, causing the second and third arguments to be passed with the same value.

In the following example, `main` passes `func` two values, 5 and 7. The function `func` receives copies of these values, and accesses them by the identifiers `a` and `b`. The function `func` changes the value of `a`. When control passes back to `main`, the actual values of `x` and `y` are not changed. The called function `func` only receives copies of `x` and `y`, and not the actual values themselves.

```
#include <stdio.h>
int main(void)
{
    int x = 5, y = 7;
    func(x, y);
    printf("In main, x = %d   y = %d\n", x, y);
}
void func (int a, int b)
{
    a +=b;
    printf("In func, a = %d   b = %d\n", a, b);
}
```

This program produces the following output:

```
In func, a = 12    b = 7
In main, x = 5    y = 7
```

### Related Reading

“Functions” on page 77  
“Function Calls” on page 72  
“Types of Expressions” on page 70  
“Operands” on page 69  
“lvalues” on page 70  
“Example of the main() Function” on page 145  
“Examples of Function Declarations” on page 139  
“Examples of Function Definitions” on page 142  
“Function Declarations” on page 138  
“Function Definitions” on page 139  
“float, double” on page 80  
“int, long, short” on page 81

---

## Implicit Type Conversions

### Integral Promotions

Certain fundamental types can be used wherever an integer can be used. The fundamental types that can be converted through integral promotion are:

- **char**
- **short int**
- enumerators
- objects of enumeration type
- integer bit fields (both signed and unsigned)

If the value cannot be represented by an **int**, the value is converted to an **unsigned int**.

**Note:** Integral promotions are not performed on **long** or **long long** integers.

### Related Reading

“Standard Type Conversions”  
“Arithmetic Conversions” on page 76  
“Arithmetic Conversions Table” on page 137  
“Arithmetic Conversions for extended Level C” on page 167  
“char” on page 79  
“int, long, short” on page 81

## Standard Type Conversions

Many C operators cause *implicit type conversions*, which change the type of a value. When you add values of operands having different data types, both values are first converted to the same type. For example, when a **short int** value and an **int** value are added together, the **short int** value is converted to the **int** type.

Implicit type conversions can occur when:

- A value is prepared for an arithmetic or logical operation.
- An assignment is made to an lvalue that has a different type than the assigned value.
- A prototyped function is provided a value that has a different type than the parameter.

- The value specified in the **return** statement of a function has a different type from the defined return type for the function.

You can perform explicit type conversions using the cast operator or the function style cast. For more information on explicit type conversions, see Type Casting (page 122).

## Signed-Integer Conversions

The compiler converts a signed integer to a shorter integer by truncating the high-order bits and converting the variable to a longer signed integer by sign-extension.

Conversion of signed integers to floating-point values takes place without loss of information, except when an **int** or **long int** value is converted to a **float**, in which case some precision may be lost. When a signed integer is converted to an unsigned integer, the signed integer is converted to the size of the unsigned integer, and the result is interpreted as an unsigned value.

## Unsigned-Integer Conversions

An unsigned integer is converted to a shorter unsigned or signed integer by truncating the high-order bits. An unsigned integer is converted to a longer unsigned or signed integer by zero-extending. Zero-extending pads the leftmost bits of the longer integer with binary zeros.

When an unsigned integer is converted to a signed integer of the same size, no change in the bit pattern occurs. However, the value changes if the sign bit is set.

## Floating-Point Conversions

A **float** value converted to a **double** undergoes no change in value. A **double** converted to a **float** is represented exactly, if possible. If the compiler cannot exactly represent the **double** value as a **float**, the value loses precision. If the value is too large to fit into a **float**, the result is undefined.

When a floating-point value is converted to an integer value, the decimal fraction portion of the floating-point value is discarded in the conversion. If the result is too large for the given integer type, the result of the conversion is undefined.

## Pointer Conversions

Pointer conversions are performed when pointers are used, including pointer assignment, initialization, and comparison.

A constant expression that evaluates to zero can be converted to a pointer. This pointer will be a null pointer (pointer with a zero value), and is guaranteed not to point to any object.

Any pointer to an object that is not a **const** or **volatile** object can be converted to a **void\***. You can also convert any pointer to a function to a **void\***, provided that a **void\*** has sufficient bits to hold it.

You can convert an expression with type array of some type to a pointer to the initial element of the array, except when the expression is used as the operand of the & (address) operator or the **sizeof** operator.

You can convert an expression with a type of function returning T to a pointer to a function returning T, except when the expression is used as the operand of the & (address) operator, the () (function call) operator, or the **sizeof** operator.

You can convert an integer value to an address offset.

For more information on pointer conversions, see Pointer Arithmetic (page 92).

## Function Argument Conversions

If no function prototype declaration is visible when a function is called, the compiler can perform default argument promotions, which consist of the following:

- Integral promotions
- Arguments with type **float** are converted to type **double**.

## Other Conversions

By definition, the **void** type has no value. Therefore, it cannot be converted to any other type, and no other value can be converted to **void** by assignment. However, a value can be explicitly cast to **void**.

No conversions between structure or union types are allowed. You can convert from an **enum** to any integral type but not from an integral type to an **enum**.

### Related Reading

“Operands” on page 69

“lvalues” on page 70

“Integral Promotions” on page 74

“Arithmetic Conversions”

“Arithmetic Conversions Table” on page 137

“Arithmetic Conversions for extended Level C” on page 167

“Data Type Qualifiers” on page 115

“int, long, short” on page 81

“float, double” on page 80

“enum” on page 82

“void” on page 85

“return” on page 154

sizeof (page 122)

## Arithmetic Conversions

Most operators perform type conversions to bring the operands of an expression to a common type or to extend short values to the integer size used in machine operations. The conversions depend on the specific operator and the type of the operand or operands. However, many operators perform similar conversions on operands of integer and floating-point types. These standard conversions are known as the *arithmetic conversions* because they apply to the types of values ordinarily used in arithmetic.

Arithmetic conversions are used for matching operands of arithmetic operators to a common type. See “Arithmetic Conversions Table” on page 137 to see how operand type mismatches are resolved.



## Related Reading

“Operands” on page 69  
“Integral Promotions” on page 74  
“Standard Type Conversions” on page 74  
“Arithmetic Conversions Table” on page 137  
“Arithmetic Conversions for extended Level C” on page 167

---

## Functions

Functions specify the logical structure of a program, and define how operations are implemented.

A function *declaration* consists of a return type, a name, and an argument list. It is used to declare the format and existence of a function prior to its use.

A function *definition* contains a function declaration, and the body of the function. A function can have only one definition.

C functions can be declared or defined in two ways:

<i>prototyped</i>	Type information is provided with each parameter. The compiler uses the function prototype for argument type checking and argument conversions. Prototypes can appear several times in a program, provided the declarations are compatible. They allow the compiler to check for mismatches between the parameters of a function call and those in the function declaration.
<i>nonprototyped</i>	No type information is provided in the function declaration. Type information for each parameter in a function definition is provided after a list of parameters.

Prototypes are the preferred style of function declaration. The ANSI C standard has declared the nonprototyped style obsolete.

## Calling Functions and Passing Arguments

A function call specifies a function name and a list of arguments. The calling function passes the value of each argument to the specified function. The argument list is surrounded by parentheses, and each argument is separated by a comma. The argument list can be empty.

The arguments to a function are evaluated before the function is called. When an argument is passed in a function call, the function receives a copy of the argument value. If the value of the argument is an address, the called function can use indirection to change the contents pointed to by the address. If a function or array is passed as an argument, the argument is converted to a pointer that points to the function or array.

Arguments passed to parameters in prototype declarations will be converted to the declared parameter type. For nonprototype function declarations, “**char**” on page 79 and “**int, long, short**” on page 81 parameters are promoted to “**int, long, short**” on page 81, and “**float, double**” on page 80 to “**float, double**” on page 80.

The order in which arguments are evaluated and passed to the function is implementation-defined.

For example, the following sequence of statements calls the function `tester`:

```
int x;  
x = 1;  
tester(x++, x);
```

The call to `tester` in the example may produce different results on different compilers. Depending on the implementation, `x++` may be evaluated first or `x` may be evaluated first. To avoid the ambiguity and have `x++` evaluated first, replace the preceding sequence of statements with the following:

```
int x, y;
x = 1;
y = x++;
tester(y, x);
```

The value of the second parameter in the following example is unpredictable:

```
int x;
x = 1;
tester(x++, x);
```

The following sequence of statements avoids this ambiguous function call by having `x++` evaluated first:

```
int x, y;
x = 1;
y = x++;
tester(y, x);
```

### Related Reading

- “Function Calls” on page 72
- “Arithmetic Conversions” on page 76
- “Arithmetic Conversions Table” on page 137
- “Integral Promotions” on page 74
- “Standard Type Conversions” on page 74
- “Function Declarations” on page 138
- “Function Definitions” on page 139
- “return” on page 154
- “main() Function” on page 144
- “Example of the main() Function” on page 145
- “Examples of Function Declarations” on page 139
- “Examples of Function Definitions” on page 142
- “char” on page 79
- “int, long, short” on page 81
- “float, double” on page 80

---

## C Language Levels

To help you avoid conflicts between the different C language definitions in existence, the C for AIX compiler supports several levels of the C language. Available language levels and their descriptions are:

Level	Description
<b>ansi</b>	Conforms to the American National Standards Institute (ANSI) C standard.
<b>classic</b>	Conforms closely to the K&R level preprocessor, enabling the compilation of many non-ANSI programs.
<b>extended</b>	High compatibility with RT C source code. <b>extended</b> level C is defined as <b>ansi</b> level C, extended for compatibility with the RT compiler. RT compatibility conflicts with the ANSI C standard under certain conditions.
<b>saal2</b>	Systems Application Architecture (SAA) CPI C Level 2. <b>saal2</b> level C conforms to SAA C with some deviations.
<b>saa</b>	The highest level SAA C definition available. This is currently SAA Level 2 C.

**Note:** **saal2** and **saa** C language levels are options of the C for AIX compiler, and are not part of the SAA Common Programming Interface.

You should use the **ansi** language level for most new programs. You can specify support for other language levels with the **-qlanglvl** compiler option if your environment and applications require it.

The C for AIX compiler also uses various compiler invocation modes to provide additional support for specific environments and levels of the C language.

### Related Reading

- “Chapter 3. Using the C for AIX Compiler” on page 5
- “Compiler Modes” on page 5
- “Invoking the Compiler” on page 8
- “Compiler Options and Their Defaults” on page 218
- “Conflicts Between extended C and Other Levels” on page 162
- “Extensions to RT C Provided by extended C” on page 164
- “Exceptions to ansi C Addressed by classic C” on page 164
- “saal2 C Deviations from SAA Level 2 C” on page 167
- “Arithmetic Conversions for extended Level C” on page 167
- “Summary of C Language Level Conflicts” on page 170
- “langlvl” on page 286

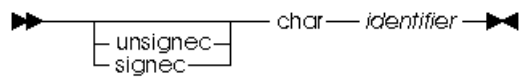
---

## Basic Data Types

### char

Specifier	Description
char	Use to declare arrays of characters, pointers to characters, and arrays of pointers to characters.
signed char, unsigned char	Use to declare numeric variables that occupy a single byte.

To declare a data object having a character type, use the **char** type specifier. The **char** specifier has the form:



The declarator for a simple character declaration is an identifier. You can initialize a simple character with a character constant or with an expression that evaluates to an integer.

The C language has three character data types: **char**, **signed char**, and **unsigned char**. These data types are not compatible with each other, but each provides enough storage to hold any member of the ASCII character set. The amount of storage allocated for a **char** is implementation-dependent. The C for AIX compiler uses 8 bits to represent a character, as defined by the **CHAR\_BIT** macro in the **<limits.h>** header.

The default character type behaves like an **unsigned char**. To change this default, use **#pragma chars** or the **-qchars** compiler option.

If it does not matter whether a **char** data object is **signed** or **unsigned**, you can declare the object as having the data type **char**. Otherwise, explicitly declare the object as **signed char** or **unsigned char**. When a **char** (**signed** or **unsigned**) is widened to an **int**, its value is preserved.

The following example defines the identifier `end_of_string` as a constant object of type **char** having the initial value `\0` (the null character):

```
const char end_of_string = '\0';
```

The following example defines the **unsigned char** variable `switches` as having the initial value 3:

```
unsigned char switches = 3;
```

The following example defines `string_pointer` as a pointer to a character:

```
char *string_pointer;
```

The following example defines `name` as a pointer to a character. After initialization, `name` points to the first letter in the character string "Johnny":

```
char *name = "Johnny";
```

The following example defines a one-dimensional array of pointers to characters. The array has three elements. Initially they are a pointer to the string "Venus", a pointer to "Jupiter", and a pointer to "Saturn":

```
static char *planets[ ] = { "Venus", "Jupiter", "Saturn" };
```

### Related Reading

"Character Constants" on page 50

"Arrays" on page 86

"Pointers" on page 90

"#pragma chars Preprocessor Directive" on page 365

"chars" on page 244

## float, double

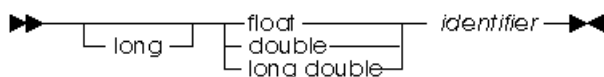
Specifier	Description
<code>float</code>	Allocates 4 bytes of data storage.
<code>double</code>	Allocates 8 bytes of data storage.
<code>long double</code>	Allocates 8 bytes of data storage in 32-bit mode, or 16 bytes if the <b>-qdbl128</b> or <b>-qlongdouble</b> option is in effect.

### Notes:

1. The amount of storage allocated for a **float**, **double**, or **long double** floating-point variable is implementation-dependent. On all compilers, the storage size of a **float** variable is less than or equal to the storage size of a **double** variable.
2. In extended mode, the C compiler supports **long float**, but this is a non-portable language extension.

To declare a data object having a floating-point type, use the *float specifier*.

The **float** specifier has the form:



The declarator for a simple floating-point declaration is an identifier. You can initialize a simple floating-point variable with a float constant or with a variable or expression that evaluates to an integer or floating-point number. The storage class of a variable determines how you initialize the variable.

The following example defines the identifier `pi` as an object of type **double**:

```
double pi;
```

The following example defines the **float** variable `real_number` with the initial value 100.55:

```
static float real_number = 100.55f;
```

The following example defines the **float** variable `float_var` with the initial value 0.0143:

```
float float_var = 1.43e-2f;
```

The following example declares the **long double** variable `maximum`:

```
extern long double maximum;
```

The following example defines the array `table` with 20 elements of type **double**:

```
double table[20];
```

### Related Reading

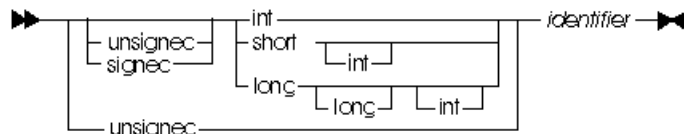
- “Floating-Point Constants” on page 49
- “Compile-Time Floating-Point Arithmetic” on page 26
- “Floating-Point Compiler Options” on page 27
- “ldbl128, longdouble” on page 289

## int, long, short

Specifier	Description
short, short int	Allocates 2 bytes of data storage.
int	Allocates 4 bytes of data storage.
long, long int	Allocates 4 bytes of data storage in 32-bit mode, 8 bytes in 64-bit mode.
long long, long long int	Allocates 8 bytes of data storage. The C compiler supports <b>long long</b> , but this is not a standard C data type. Though needed for some AIX system programming, it may not be portable to other systems.
<b>Notes:</b> The amount of storage allocated for an <b>int</b> , <b>short</b> , or <b>long</b> integer variable is implementation-dependent.	

To declare a data object having an integer data type, use an **int** type specifier.

The **int** specifier has the form:



The declarator for a simple integer definition or declaration is an identifier. You can initialize a simple integer definition with an integer constant or with an expression that evaluates to a value that can be assigned to an integer. The storage class of a variable determines how you can initialize the variable.

The **unsigned** prefix indicates that the object is a nonnegative integer. Each unsigned type provides the same size storage as its signed equivalent. For example, **int** reserves the same storage as **unsigned int**. Because a signed type reserves a sign bit, an unsigned type can hold a larger positive integer than the equivalent signed type.

The following example defines the **short int** variable `flag`:

```
short int flag;
```

The following example defines the **int** variable `result`:

```
int result;
```

The following example defines the **unsigned long int** variable `ss_number` as having the initial value 438888834:

```
unsigned long ss_number = 438888834ul;
```

The following example defines the identifier `sum` as an object of type **int**. The initial value of `sum` is the result of the expression `a + b`:

```
extern int a, b;  
auto sum = a + b;
```

### Related Reading

“Integer Constant” on page 48

## enum

An *enumeration* data type represents a set of values that you declare. You can define an enumeration data type and all variables that have that enumeration type in one statement, or you can separate the declaration of the enumeration data type from all variable definitions. The identifier associated with the data type (not an object) is called an *enumeration tag*.

The diagram shows the syntax for an enumeration declaration: `enum` followed by an *identifier* in brackets, then an opening curly brace, followed by one or more *enumerator* identifiers in single quotes, a closing curly brace, a semicolon, and a closing arrowhead.

*identifier* Names the data type (like the tag on a **struct** data type).  
*enumerator* Provides the data type with a set of values.

The diagram shows the syntax for an enumerator declaration: an *identifier* in brackets, followed by an equals sign, an *integral\_constant\_expression* in brackets, and a closing arrowhead.

Each enumerator constant in the list has its own *identifier*, and represents an integer value. The integer value of an enumerator can be set implicitly by the position of the enumerator within the list, or explicitly by assigning an *integral\_constant\_expression* value to that enumerator.

To conserve space, enumerations may be stored in spaces smaller than that of an **int**.

### Enumeration Constants

When you define an enumeration data type, you specify a set of identifiers that the data type represents. Each identifier in this set is called an *enumeration constant*.

The value of the constant is determined in the following way:

1. An equal sign (=) and a constant expression after the enumeration constant gives an explicit value to the constant. The identifier represents the value of the constant expression.
2. If no explicit value is assigned, the leftmost constant in the list receives the value zero (0).
3. Identifiers with no explicitly assigned values receive the integer value that is one greater than the value represented by the previous identifier.

Each enumeration constant has an integer value. Use an enumeration constant anywhere an integer constant is allowed.

Each enumeration constant must be unique within the scope in which the enumeration is defined. In the following example, the declarations of `average` on line 4 and of `poor` on line 5 cause compiler error messages:

```

1 func()
2 {
3     enum score { poor, average, good };
4     enum rating { below, average, above };
5     int poor;
6 }

```

### Defining Enumeration Variables

An enumeration variable definition contains an optional storage class specifier, a type specifier, a declarator, and an optional initializer. The type specifier contains the keyword **enum** followed by the name of the enumeration data type. You must declare the enumeration data type before you can define a variable having that type.

The initializer for an enumeration variable contains the = symbol followed by an expression. The initializer expression must evaluate to an **int** value.

The first line of the following example declares the enumeration tag `grain`. The second line defines the variable `g_food` and gives variable `g_food` the initial value of `barley` (2).

```

enum grain { oats, wheat, barley, corn, rice };
enum grain g_food = barley;

```

The type specifier `enum grain` indicates that the value of `g_food` is a member of the enumerated data type `grain`.

### Defining an Enumeration Type and Enumeration Objects in the Same Statement

You can define a type and a variable in one statement by using a declarator and an optional initializer after the type definition. To specify a storage class specifier for the variable, you must put the storage class specifier at the beginning of the declaration. For example:

```

register enum score { poor=1, average, good } rating = good;

```

This example is equivalent to the following two declarations:

```

enum score { poor=1, average, good };
register enum score rating = good;

```

Both examples define the enumeration data type `score` and the variable `rating`. `rating` has the storage class specifier **register**, the data type `enum score`, and the initial value 3 (or `good`).

Combining a data type definition with the definitions of all variables having that data type lets you leave the data type unnamed. For example:

```

enum { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
      Saturday } weekday;

```

defines the variable `weekday`, which can be assigned any of the specified enumeration constants.

### Related Reading

“Identifiers” on page 47

“Constant Expressions” on page 71

“Examples of Enumerator Declaration and Use”

### Examples of Enumerator Declaration and Use

The following data type declarations list `oats`, `wheat`, `barley`, `corn`, and `rice` as enumeration constants. The number under each constant shows the integer value.

```

enum grain { oats, wheat, barley, corn, rice };
/*      0      1      2      3      4      */
enum grain { oats=1, wheat, barley, corn, rice };
/*      1      2      3      4      5      */
enum grain { oats, wheat=10, barley, corn=20, rice };
/*      0      10      11      20      21      */

```

It is possible to associate the same integer with two different enumeration constants. For example, the following definition is valid. The identifiers `suspend` and `hold` have the same integer value.

```

enum status { run, clear=5, suspend, resume, hold=6 };
/*      0      5      6      7      6      */

```

The following example is a different declaration of the enumeration tag `status`:

```

enum status { run, create, clear=5, suspend };
/*      0      1      5      6      */

```

The following program receives an integer as input. The output is a sentence that gives the French name for the weekday that is associated with the integer. If the integer is not associated with a weekday, the program prints "C'est le mauvais jour."

```

** Example program using enumerations
**/
#include <stdio.h>
enum days {
    Monday=1, Tuesday, Wednesday,
    Thursday, Friday, Saturday, Sunday
} weekday;
void french(enum days);
int main(void)
{
    int num;
    printf("Enter an integer for the day of the week. "
           "Mon=1,...,Sun=7\n");
    scanf("%d", &num);
    weekday=num;
    french(weekday);
    return(0);
}
void french(enum days weekday)
{
    switch (weekday)
    {
        case Monday:
            printf("Le jour de la semaine est lundi.\n");
            break;
        case Tuesday:
            printf("Le jour de la semaine est mardi.\n");
            break;
        case Wednesday:
            printf("Le jour de la semaine est mercredi.\n");
            break;
        case Thursday:
            printf("Le jour de la semaine est jeudi.\n");
            break;
        case Friday:
            printf("Le jour de la semaine est vendredi.\n");
            break;
        case Saturday:
            printf("Le jour de la semaine est samedi.\n");
            break;
        case Sunday:
            printf("Le jour de la semaine est dimanche.\n");
            break;
    }
}

```



```

        default:
            printf("C'est le mauvais jour.\n");
    }
}

```

### Related Reading

“enum” on page 82

## void

The **void** data type always represents an empty set of values. The only object that can be declared with the type specifier **void** is a pointer.

When a function does not return a value, you should use **void** as the type specifier in the function definition and declaration. An argument list for a function taking no arguments is **void**.

You cannot declare a variable of type **void**, but you can explicitly convert any expression to type **void** with the resulting expression used only as one of the following:

- An expression statement
- The left operand of a comma expression
- The second or third operand in a conditional expression.

### Related Reading

“Example of a void Declaration”

## Example of a void Declaration

On line 7 of the following example, the function `find_max` is declared as having type **void**. Lines 15 through 26 contain the complete definition of `find_max`.

**Note:** The use of the **sizeof** operator in line 13 is a standard method of determining the number of elements in an array.

```

1  /**
2  ** Example of void type
3  **/
4  #include <stdio.h>
5
6  /* declaration of function find_max */
7  extern void find_max(int x[ ], int j);
8
9  int main(void)
10 {
11     static int numbers[ ] = { 99, 54, -102, 89 };
12     find_max(numbers, (sizeof(numbers) / sizeof(numbers[0])));
13
14     return(0);
15 }
16
17
18 void find_max(int x[ ], int j)
19 { /* begin definition of function find_max */
20     int i, temp = x[0];
21
22     for (i = 1; i < j; i++)
23     {
24         if (x[i] > temp)

```

```

25         temp = x[i];
26     }
27     printf("max number = %d\n", temp);
28 } /* end definition of function find_max */

```

### Related Reading

“void” on page 85

---

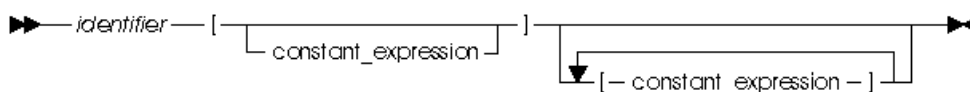
## Derived Data Types

### Arrays

An *array* is an ordered group of data objects. Each object is called an *element*. All elements within an array have the same data type.

Use any type specifier in an array definition or declaration. Array elements can be of any data type, except function. You can, however, declare an array of pointers to functions.

#### Declaring an Array



*identifier*                      The name of the array. If preceded by an \* (asterisk), the array is an array of pointers.  
*constant expression*        Positive integer expression describing the number of elements in a given dimension of the array. An array can have more than one dimension.

The following example defines a one-dimensional array that contains four elements having type **char**:

```
char list[4];
```

The first subscript of each dimension is 0. The array `list` contains the elements:

```
list[0]
list[1]
list[2]
list[3]
```

The following example defines a two-dimensional array that contains six elements of type **int**:

```
int roster[3][2];
```

Multidimensional arrays are stored in row-major order. When elements are referred to in order of increasing storage location, the last subscript varies the fastest. For example, the elements of array `roster` are stored in the order:

```
roster[0][0]
roster[0][1]
roster[1][0]
roster[1][1]
roster[2][0]
roster[2][1]
```

You can leave the first (and only the first) set of subscript brackets empty in

- Array definitions that contain initializations
- **extern** declarations

- Parameter declarations.

In array definitions that leave the first set of subscript brackets empty, the initializer determines the number of elements in the first dimension. In a one-dimensional array, the number of initialized elements becomes the total number of elements. In a multidimensional array, the initializer is compared to the subscript declarator to determine the number of elements in the first dimension.

An unsubscripted array (for example, `region` instead of `region[4]`) represents a pointer whose value is the address of the first element of the array, provided the array has been previously declared. An unsubscripted array name with square brackets (for example, `region[]`) is allowed only when declaring arrays at file scope or in the argument list of a function declaration. In declarations, only the first dimension can be left empty, and you must specify the sizes of any additional dimensions declared.

Whenever an array is used in a context (such as a parameter) where it cannot be used as an array, the identifier is treated as a pointer. The only exceptions are when an array is used as an operand to the **sizeof** expression or with an address (&) operator.

### Initializing Arrays

The initializer for an array contains the = symbol followed by a comma-separated list of constant expressions enclosed in braces ({ }). You do not need to initialize all elements in an array. Elements that are not initialized (in **extern** and **static** definitions only) receive the value 0 of the appropriate type. You cannot have more initializers than the number of elements in the array.

The initializer must be a constant expression if the structure has **static** storage duration or if you are compiling your source code in **ansi** mode.

**Note:** Array initializations can be either *fully braced* (with braces around each dimension) or *unbraced* (with only one set of braces enclosing the entire set of initializers). Avoid placing braces around some dimensions and not around others.

### Initializing a one-dimensional character array

Initialize a one-dimensional character array by specifying:

- A brace-enclosed comma-separated list of constants, each of which can be contained in a character
- A string constant. (Braces surrounding the constant are optional.)

Initializing a string constant places the null character (\0) at the end of the string if there is room or if the array dimensions are not specified.

### Initializing a multidimensional array

Initialize a multidimensional array by:

- Listing the values of all elements you want to initialize, in the order that the compiler assigns the values. The compiler assigns values by increasing the subscript of the last dimension fastest. This form of a multidimensional array initialization looks like a one-dimensional array initialization. The following definition completely initializes the array `month_days`:

```
static month_days[2][12] =
{
    31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,
    31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};
```

- Using braces to group the values of the elements you want initialized. You can put braces around each element, or around any nesting level of elements. The following definition contains two elements in the first dimension. (You can consider these elements as rows.) The initialization contains braces around each of these two elements:

```

static int month_days[2][12] =
{
    { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
    { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
};

```

- Using use nested braces to initialize dimensions and elements in a dimension selectively.

### Related Reading

“String Literals” on page 51  
 “Declarators” on page 64  
 “Initializers” on page 65  
 “Examples of Array Declaration and Use”  
 Array Subscript (page 118)  
 “Pointers” on page 90

## Examples of Array Declaration and Use

The following show four different character array initializations:

```

static char name1[] = { 'J', 'a', 'n' };
static char name2[] = { "Jan" };
static char name3[3] = "Jan";
static char name4[4] = "Jan";

```

These initializations create the following elements:

name1		name2		name3		name4	
Element	Value	Element	Value	Element	Value	Element	Value
name1[0]	J	name2[0]	J	name3[0]	J	name4[0]	J
name1[1]	a	name2[1]	a	name3[1]	a	name4[1]	a
name1[2]	n	name2[2]	n	name3[2]	n	name4[2]	n
		name2[3]	\0			name4[3]	\0

Note that the NULL character (\0) is lost for name1[] and name3[3]. A compiler warning is issued for name3[3].

The following program defines a floating-point array called prices.

The first **for** statement prints the values of the elements of prices. The second for statement adds five percent to the value of each element of prices, and assigns the result to total, and prints the value of total.

```

/**
 ** Example of one-dimensional arrays
 **/
#include <stdio.h>
#define ARR_SIZE 5
int main(void)
{
    static float const prices[ARR_SIZE] = { 1.41, 1.50, 3.75, 5.00, .86 };
    auto float total;
    int i;
    for (i = 0; i < ARR_SIZE; i++)
    {
        printf("price = $%.2f\n", prices[i]);
    }
    printf("\n");
    for (i = 0; i < ARR_SIZE; i++)
    {
        total = prices[i] * 1.05;
    }
}

```

```

    printf("total = $%.2f\n", total);
}
return(0);
}

```

This program produces the following output:

```

price = $1.41
price = $1.50
price = $3.75
price = $5.00
price = $0.86
total = $1.48
total = $1.57
total = $3.94
total = $5.25
total = $0.90

```

The following program defines the multidimensional array `salary_tbl`. A `for` loop prints the values of `salary_tbl`.

```

/**
 ** Example of a multidimensional array
 **/
#include <stdio.h>
#define ROW_SIZE 3
#define COLUMN_SIZE 5
int main(void)
{
    static int salary_tbl[ROW_SIZE][COLUMN_SIZE] =
    {
        { 500, 550, 600, 650, 700 },
        { 600, 670, 740, 810, 880 },
        { 740, 840, 940, 1040, 1140 }
    };
    int grade, step;
    for (grade = 0; grade < ROW_SIZE; ++grade)
        for (step = 0; step < COLUMN_SIZE; ++step)
        {
            printf("salary_tbl[%d] [%d] = %d\n", grade, step,
                salary_tbl[grade][step]);
        }
    return(0);
}

```

This program produces the following output:

```

salary_tbl[0] [0] = 500
salary_tbl[0] [1] = 550
salary_tbl[0] [2] = 600
salary_tbl[0] [3] = 650
salary_tbl[0] [4] = 700
salary_tbl[1] [0] = 600
salary_tbl[1] [1] = 670
salary_tbl[1] [2] = 740
salary_tbl[1] [3] = 810
salary_tbl[1] [4] = 880
salary_tbl[2] [0] = 740
salary_tbl[2] [1] = 840
salary_tbl[2] [2] = 940
salary_tbl[2] [3] = 1040
salary_tbl[2] [4] = 1140

```

### Related Reading

“Arrays” on page 86

## Pointers

A *pointer* type variable holds the address of a data object or function. A pointer can refer to an object of any one data type, but cannot point to a bit field or to an object having the **register** storage class specifier. Some common uses for pointers are:

- To access dynamic data structures such as linked lists, trees, and queues.
- To access elements of an array or members of a structure.
- To access an array of characters as a string.
- To pass *by reference* the address of a variable to a function. By referencing a variable through its address, a function can change the contents of that variable.

Pointers occupy 4 bytes of data storage in 32-bit mode, and 8 bytes in 64-bit mode.

### Declaring a Pointer

A pointer is declared by placing an \* (asterisk) after the data type specifier and before the identifier. The following example declares `pcoat` as a pointer to an object having type **long**:

```
extern long *pcoat;
```

If the keyword **volatile** appears before the \*, the declarator describes a pointer to a **volatile** object. If the keyword **volatile** comes between the \* and the identifier, the declarator describes a **volatile** pointer. The keyword **const** operates in the same manner as the **volatile** keyword described. In the following example, `pvolt` is a constant pointer to an object having type **short**:

```
short * const pvolt;
```

The following example declares `pnut` as a pointer to an **int** object having the **volatile** qualifier:

```
extern int volatile *pnut;
```

The following example defines `psoup` as a **volatile** pointer to an object having type **float**:

```
float * volatile psoup;
```

The following example defines `pfowl` as a pointer to an enumeration object of type `bird`:

```
enum bird *pfowl;
```

The next example declares `pvish` as a pointer to a function that takes no parameters and returns a **char** object:

```
char (*pvish)(void);
```

### Assigning Pointers

When you use pointers in an assignment operation, you must ensure that the types of the pointers in the operation are compatible.

The following example shows compatible declarations for the assignment operation:

```
float subtotal;  
float * sub_ptr;  
.  
:  
.  
sub_ptr = &subtotal;  
printf("The subtotal is %f\n", *sub_ptr);
```

The next example shows incompatible declarations for the assignment operation:

```
double league;
int * minor;
    .
    .
    .
minor = &league;    /* error */
```

### Initializing Pointers

The initializer is an = (equal sign) followed by the expression that represents the address that the pointer is to contain. The following example defines the variables `time` and `speed` as having type **double** and `amount` as having type pointer to a **double**. The pointer `amount` is initialized to point to `total`:

```
double total, speed, *amount = &total;
```

The compiler converts an unsubscripted array name to a pointer to the first element in the array. You can assign the address of the first element of an array to a pointer by specifying the name of the array. The following two sets of definitions are equivalent. Both define the pointer `student` and initialize `student` to the address of the first element in `section`:

```
int section[80];
int *student = section;
```

is equivalent to:

```
int section[80];
int *student = &section[0];
```

You can assign the address of the first character in a string constant to a pointer by specifying the string constant in the initializer.

The following example defines the pointer variable `string` and the string constant "abcd". The pointer `string` is initialized to point to the character `a` in the string "abcd".

```
char *string = "abcd";
```

The following example defines `weekdays` as an array of pointers to string constants. Each element points to a different string. The pointer `weekdays[2]`, for example, points to the string "Tuesday".

```
static char *weekdays[ ] =
{
    "Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"
};
```

A pointer can also be initialized to `NULL` using any integer constant expression that evaluates to 0, for example `char * a=0;`. Such a pointer is a *NULL pointer*. It does not point to any object.

### Using Pointers

Two operators are commonly used in working with pointers, the address (&) operator and the indirection (\*) operator. You can use the & operator to refer to the address of an object. For example, the following statement assigns the address of `x` to the variable `p_to_x`. The variable `p_to_x` has been defined as a pointer.

```
int x, *p_to_x;
p_to_x = &x;
```

The \* (indirection) operator lets you access the value of the object a pointer refers to. The following statement assigns to `y` the value of the object that `p_to_x` points to:

```
float y, *p_to_x;
.
.
.
y = *p_to_x;
```

The following statement assigns the value of `y` to the variable that `*p_to_x` references:

```
char y ,
     *p_to_x,
.
.
.
*p_to_x = y;
```

You cannot use pointers to reference bit fields or objects having the **register** storage class specifier.

### Pointer Arithmetic

You can perform a limited number of arithmetic operations on pointers. These operations are:

- Increment and decrement
- Addition and subtraction
- Comparison
- Assignment

The increment (`++`) operator increases the value of a pointer by the size of the data object the pointer refers to. For example, if the pointer refers to the second element in an array, the `++` makes the pointer refer to the third element in the array.

The decrement (`--`) operator decreases the value of a pointer by the size of the data object the pointer refers to. For example, if the pointer refers to the second element in an array, the `--` makes the pointer refer to the first element in the array.

You can add a pointer to an integer, but you cannot add a pointer to a pointer.

If the pointer `p` points to the first element in an array, the following expression causes the pointer to point to the third element in the same array:

```
p = p + 2;
```

If you have two pointers that point to the same array, you can subtract one pointer from the other. This operation yields the number of elements in the array that separate the two addresses that the pointers refer to.

You can compare two pointers with the following operators: `==`, `!=`, `<`, `>`, `<=`, and `>=`.

Pointer comparisons are defined only when the pointers point to elements of the same array. Pointer comparisons using the `==` and `!=` operators can be performed even when the pointers point to elements of different arrays.

You can assign to a pointer the address of a data object, the value of another compatible pointer or the `NULL` pointer.

### Passing Pointer Values to Functions

Pointers allow a called function to alter the value of a variable in the calling function. Any changes to a



variable passed as an argument to a called function are not returned to the calling function. However, if a pointer to a variable is passed as an argument, the called function can alter the value of the variable the pointer refers to.

### Related Reading

“Declarators” on page 64  
“Initializers” on page 65  
“Examples of Pointer Declaration and Use”  
Address (&) Operator (page 121)  
Indirection (\*) Operator (page 122)  
“Data Type Qualifiers” on page 115  
“char” on page 79  
“float, double” on page 80  
“float, double” on page 80  
“int, long, short” on page 81  
“register” on page 111  
“Data Type Qualifiers” on page 115  
“ldbl128, longdouble” on page 289

## Examples of Pointer Declaration and Use

The following program shows how you can pass a pointer to a function and change the value of the object the pointer points to:

```
/******  
** This program accepts a value for a timer, then decreases **  
** this timer value by one each time the function count_down **  
** is called. **  
*****/  
#include <stdio.h>  
int count_down(int *timer)  
int main(void)  
{  
    int t_timer;          /* local storage */  
    printf("Set timer to: _ \n");  
    scanf("%d", &t_timer);  
    if (t_timer <= 0)  
        printf("Timer was set to a negative value\n");  
    else  
    {  
        while ( count_down(&t_timer) ) /* while timer not zero */  
        {  
            printf("Timer still counting. %d\n", t_timer);  
        }  
        printf("Timer has reached zero.\n");  
    }  
} /* End main */ </pre>  
  
/******  
** This function decreases the value of timer by decrements **  
** of 1 and returns false when the timer reaches zero. **  
*****/  
int count_down(int *timer) /* receives a copy of a pointer to  
                           t_timer */  
{  
    return(--*timer);      /* modifying t_timer in main */  
} /* End count_down */
```

Interaction with this program could produce the following sessions:

```

Output      Set timer to: _
Input       6
Output      Timer still counting. 5
            Timer still counting. 4
            Timer still counting. 3
            Timer still counting. 2
            Timer still counting. 1
            Timer has reached zero.

```

The following program contains pointer arrays:

```

/*****
** Program to search for the first occurrence of a specified **
** character string in an array of character strings.      **
*****/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define SIZE 20
#define EXIT_FAILURE 999
int main(void)
{
    static char *names[ ] = { "Jim", "Amy", "Mark", "Sue", NULL };
    char * find_name(char **, char *);
    char new_name[SIZE], *name_pointer;
    printf("Enter name to be searched.\n");
    scanf("%s", new_name);
    name_pointer = find_name(names, new_name);
    printf("name %s%sfound\n", new_name,
        (name_pointer == NULL) ? " not " : " ");
    exit(EXIT_FAILURE);
} /* End of main */

/*****
** Function find_name. This function searches an array **
** of names to see if a given name already exists in the **
** array. It returns a pointer to the name or NULL if **
** the name is not found. **
** **
** char **array is a pointer to arrays of pointers whose **
** names already exist. **
** **
** char *strng is a pointer to character array entered **
*****/
char * find_name(char **array, char *strng)
{
    for (; *array != NULL; array++) /* for each name */
    {
        if (strcmp(*array, strng) == 0) /* if strings match */
            return(*array); /* found it! */
    }
    return(*array); /* return the pointer */
} /* End of find_name */

```

Interaction with this program could produce the following sessions:

```

Output      Enter name to be searched.
Input       Mark
Output      name Mark found

```

or:

```

Output      Enter name to be searched._
Input       Deborah
Output      name Deborah not found

```

## Related Reading

“Pointers” on page 90

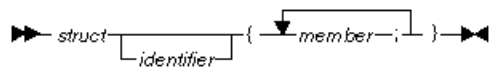
## struct (Structures)

A *structure* contains an ordered group of data objects. Unlike the elements of an array, the data objects within a structure can have varied types. Each data object within a structure is called a *member* or *field*.

Use structures to group logically-related objects. For example, to allocate storage for the components of one address, define the following variables:

```
int street_no;
char *street_name;
char *city;
char *prov;
char *postal_code;
```

### Declaring a Structure



*identifier*

Provides a tag name for the structure. If specified, subsequent declarations (in the same scope) of variables using the structure can be made by referring to the tag name. If not specified, you must place all variable definitions that refer to the structure within the declaration of the data type.

*member*

The list of members provides the data type with a description of the values that can be stored in the structure.



A member that does not represent a bit field can be of any data type and can have the **volatile** or **const** qualifier.

If a **:** (colon) and a constant expression follow the member declarator, the member represents a *bit field*. Bit fields are described in Declaring and Using Bit Fields in Structures (page 97).

A structure type declaration describes the members that are part of the structure.

Identifiers used as structure or member names can be redefined to represent different objects in the same scope without conflicting. You cannot use the name of a member more than once in a structure type, but you can use the same member name in another structure type that is defined within the same scope.

You cannot declare a structure type that contains itself as a member, but you can declare a structure type that contains a pointer to itself as a member.

### Defining a Structure Variable

A structure variable definition contains an optional storage class keyword, the **struct** keyword, a structure tag, a declarator, and an optional identifier. The structure tag indicates the data type of the structure variable.

You can declare structures having any storage class. Most compilers, however, treat structures declared with the **register** storage class specifier as automatic structures.

### Initializing Structures

The initializer contains an = (equal sign) followed by a brace-enclosed comma-separated list of values. You do not have to initialize all members of a structure.

The following definition shows a completely initialized structure:

```
struct address {
    int street_no;
    char *street_name;
    char *city;
    char *prov;
    char *postal_code;
};
static struct address perm_address =
    { 3, "Savona Dr.", "Dundas", "Ontario", "L4B 2A1"};
```

The values of perm\_address are:

Member	Value
perm_address.street_no	3
perm_address.street_name	address of string "Savona Dr."
perm_address.city	address of string "Dundas"
perm_address.prov	address of string "Ontario"
perm_address.postal_code	address of string "L4B 2A1"

The following definition shows a partially initialized structure:

```
struct address {
    int street_no;
    char *street_name;
    char *city;
    char *prov;
    char *postal_code;
};
struct address temp_address =
    { 44, "Knyvet Ave.", "Hamilton", "Ontario" };
```

The values of temp\_address are:

Member	Value
temp_address.street_no	44
temp_address.street_name	address of string "Knyvet Ave."
temp_address.city	address of string "Hamilton"
temp_address.prov	address of string "Ontario"
temp_address.postal_code	value depends on the storage class.

**Note:** The initial value of uninitialized structure members like temp\_address.postal\_code depends on the storage class associated with the member.

### Alignment of Structures

Structures are aligned according to the setting of the **-qalign** compiler option, which specifies the alignment rules the compiler uses when laying out memory storage for structures and unions. The mapping of a structure is based on the alignment setting in effect at the beginning of the structure definition.

Structures and unions with identical members, but using different alignment, are not type compatible and cannot be assigned to each other. Use the **-qextchk** compiler option to check for alignment mismatches, and refer to the attribute section of the compiler listing to find the variables that have different alignment settings.

Your code should not depend on the offset or alignment of members within a structure. Use the **offsetof** macro, defined in the `/usr/include/stddef.h` header file, to determine the offset of members in a macro. This macro is described in the *AIX Version 4 Files Reference*.

### Declaring Structure Types and Variables

To define a structure type and a structure variable in one statement, put a declarator and an optional initializer after the type definition. To specify a storage class specifier for the variable, you must put the storage class specifier at the beginning of the statement.

For example:

```
static struct {
    int street_no;
    char *street_name;
    char *city;
    char *prov;
    char *postal_code;
} perm_address, temp_address;
```

Because this example does not name the structure data type, `perm_address` and `temp_address` are the only structure variables that will have this data type. Putting an identifier after **struct**, lets you make additional variable definitions of this data type later in the program.

The structure type (or tag) cannot have the **volatile** qualifier, but a member or a structure variable can be defined as having the **volatile** qualifier.

For example:

```
static struct class1 {
    char descript[20];
    volatile long code;
    short complete;
} volatile file1, file2;
struct class1 subfile;
```

This example qualifies the structures `file1` and `file2`, and the structure member `subfile.code` as **volatile**.

**Declaring and Using Bit Fields in Structures** A structure can contain *bit fields* that allow you to access individual bits. You can use bit fields for data that requires just a few bits of storage. A bit field declaration contains a type specifier followed by an optional declarator, a colon, a constant expression, and a semicolon.

► type\_specifier — declarator — constant\_expression — ; ►

The *constant expression* specifies how many bits the field reserves.

Bit fields with a length of 0 must be unnamed. Unnamed bit fields cannot be referenced or initialized. A zero-width bit field causes the next field to be aligned on the next container boundary, where the container is the same size as the underlying type as the bit field.

The maximum bit-field length is implementation dependent. The maximum bit field length for the C for AIX compiler is 32 bits (4 bytes, or 1 word).

For portability, do not use bit fields greater than 32 bits in size.

The following restrictions apply to bit fields. You cannot:

- Define an array of bit fields
- Take the address of a bit field
- Have a pointer to a bit field

In C, you can declare a bit field as type **int**, **signed int**, or **unsigned int**. Bit fields of the type **int** are equivalent to those of type **unsigned int**.

The default integer type for a bit field is **unsigned**. Use the **bitfields=signed** option to change this default.

In extended mode C, bit fields can be any integral type. For example,

```
struct S {
    short x : 4;
    long y : 10;
    char z : 7;
} s;
```

Non-integral bit fields in extended mode C are converted to type **unsigned int** and a warning is issued. In other modes, the use of non-integral bit fields results in an error.

In **ansi** mode C, bit fields of type **unsigned char** or **unsigned short** are changed to **unsigned int**. An **unsigned short** bit field occupies 32 bits.

A bit field cannot have the **volatile** or **const** qualifier.

The following structure has three bit-field members `kingdom`, `phylum`, and `genus`, occupying 12, 6, and 2 bits respectively:

```
struct taxonomy {
    int kingdom : 12;
    int phylum : 6;
    int genus : 2;
};
```

### Alignment of Bit Fields in Structures

Bit fields are word-aligned but packed as closely as possible into the current word. The first bit field in a sequence of bit fields starts on a word boundary. For example, a structure containing only bit fields is word-aligned, but after the first bit field, the bit fields themselves do not have to begin on word boundaries. Word alignment is the default and is equivalent to setting the **-qalign=power** compiler option.

If a series of bit fields does not add up to the size of an **int**, padding can take place. The amount of padding is determined by the alignment characteristics of the members of the structure. Bit fields cannot cross word boundaries but are forced to start at the next word boundary. Alignment of structures is described in .

The following example declares the identifier `kitchen` to be of type `struct on_off`:

```
struct on_off {
    unsigned light : 1;
    unsigned toaster : 1;
    int count;          /* 4 bytes */
    unsigned ac : 4;
    unsigned : 4;
    unsigned clock : 1;
    unsigned : 0;
    unsigned flag : 1;
} kitchen ;
```

The structure `kitchen` contains eight members totalling 16 bytes. The following table describes the storage that each member occupies:

Member Name	Storage Occupied
<code>light</code>	1 bit
<code>toaster</code>	1 bit
(padding, 30 bits)	to next <b>int</b> boundary
<code>count</code>	the size of an <b>int</b>
<code>ac</code>	4 bits
(unnamed field)	4 bits
<code>clock</code>	1 bit
(padding, 23 bits)	to next <b>int</b> boundary (unnamed field)
<code>flag</code>	1 bit
(padding, 31 bits)	to next <b>int</b> boundary

All references to structure fields must be fully qualified. For instance, you cannot reference the second field by `toaster`. You must reference this field by `kitchen.toaster`.

The following expression sets the `light` field to 1:

```
kitchen.light = 1;
```

When you assign to a bit field a value that is out of its range, the bit pattern is preserved and the appropriate bits are assigned. The following expression sets the `toaster` field of the `kitchen` structure to 0 because only the least significant bit is assigned to the `toaster` field:

```
kitchen.toaster = 2;
```

### Bit Fields under the `align` Compiler Option

Bit fields are also subject to the `-qalign` compiler option.

The default alignment is `-qalign=power`. When it is in effect, bit fields are aligned as described in Alignment of Bit Fields in Structures. Bit fields have the following alignment properties under the `twobyte` and `packed` suboptions.

## twobyte

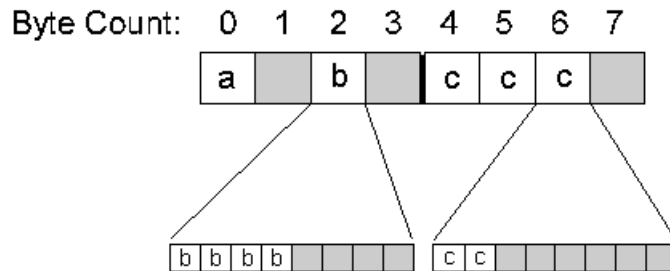
Bit fields are packed into a word and are aligned on a halfword boundary. Bit fields cannot cross word boundaries but are forced to start at the next halfword boundary even if they start on a halfword boundary.

A bit field with a width of 0 (zero) forces the next member to start at the next halfword boundary even if it is not a bit field and even if the zero-width bit field is already at a halfword boundary. A structure containing nothing but zero-width bit fields has a length equal to twice the number of zero-width bit fields.

In the following example, the bit fields in the structure `species` are aligned according to the `-qalign=twobyte` option:

```
#pragma options align=twobyte
struct species {
    char a;
    int : 0;
    int b : 4;
    int c : 18; /* 8 + 8 + 2 bits */
};
```

The following figure shows the layout of `species`. The shaded areas are padding.



Bit field `b` starts on a halfword boundary because of the unnamed zero-width `int` bit field. It occupies the first 4 bits of the third byte (byte 2 in the figure.) Because bit field `c` is larger than 2 bytes, it cannot cross the word boundary between bytes 3 and 4, but is forced to start at byte 4. It occupies bytes 4 and 5 (the first two bytes of the second word) and 2 bits of byte 6.



## packed

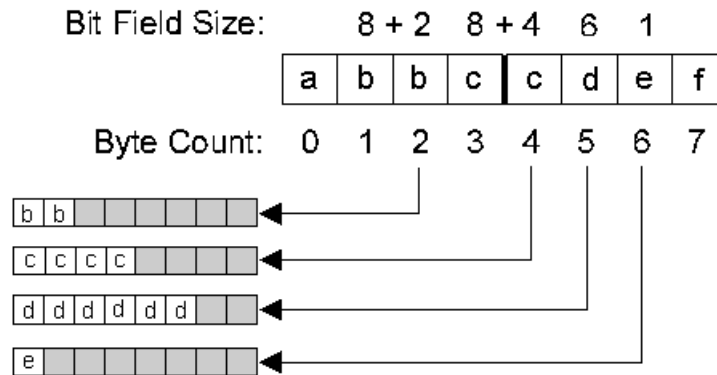
Bit fields are packed into a 1 byte space. Bit fields that cross byte boundaries are forced to start at the next available byte boundary.

A bit field with a width of 0 (zero) forces the next member to start at the next byte boundary. If the zero-width bit field is already at a byte boundary, the next structure member starts there. A non-bit field member following a bit field is aligned on the next byte boundary.

In the following example, the bit fields in the structure order are aligned according to the **-qalign=packed** option:

```
#pragma options align=packed
struct order {
    char a;
    int b : 10;
    int c : 12;
    int d : 6;
    int : 0;
    int e : 1;
    char f;
};
```

The following figure shows the layout of order. The shaded areas are padding.



Because bit field c is longer than 1 byte and cannot straddle the boundary between bytes 2 and 3, it must start at byte 3. Likewise, field d cannot cross the byte boundary between bytes 4 and 5; it is forced to start at byte 5. The zero-width bit field between field d and field e forces bit field e to start at byte 6.

## Related Reading

“Declarators” on page 64

“Initializers” on page 65

“Examples of Structure Declaration and Use”

“Incomplete Types” on page 106

Structure and Union Member Specification (page 119)

“Data Type Qualifiers” on page 115

“char” on page 79

“int, long, short” on page 81

“align” on page 234

“extchk” on page 258

## Examples of Structure Declaration and Use

The following program finds the sum of the integer numbers in a linked list:

```
/**
 ** Example program illustrating structures using linked lists
 **/
#include <stdio.h>
struct record {
```

```

        int number;
        struct record *next_num;
    };
int main(void)
{
    struct record name1, name2, name3;
    struct record *recd_pointer = &name1;
    int sum = 0;
    name1.number = 144;
    name2.number = 203;
    name3.number = 488;
    name1.next_num = &name2;
    name2.next_num = &name3;
    name3.next_num = NULL;
    while (recd_pointer != NULL)
    {
        sum += recd_pointer->number;
        recd_pointer = recd_pointer->next_num;
    }
    printf("Sum = %d\n", sum);
    return(0);
}

```

The structure type record contains two members: the integer number and next\_num, which is a pointer to a structure variable of type record.

The record type variables name1, name2, and name3 are assigned the following values:

Member	Value
name1.number	144
name1.next_num	address of name2
name2.number	203
name2.next_num	address of name3
name3.number	488
name3.next_num	NULL (indicating the end of the linked list)

The variable recd\_pointer is a pointer to a structure of type record. It is initialized to the address of name1 (the beginning of the linked list).

The **while** loop causes the linked list to be scanned until recd\_pointer equals NULL. The statement:

```
recd_pointer = recd_pointer->next_num;
```

advances the pointer to the next object in the list.

The following example shows how to define and initialize a structure within a structure.

```

struct client {
    char *name;
    struct info {
        int age;
        int weight;
    } pers_info;
} child = { "Bob", { 3, 31 } }; /* initialization */

```

### Related Reading

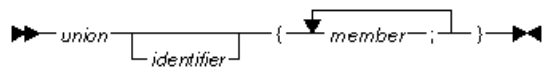
"struct (Structures)" on page 95

## union (Unions)

A *union* is an object that can hold any one of a set of named members. The members of the named set can be of any data type. Members are overlaid in storage.

The storage allocated for a union is the storage required for the largest member of the union, plus any padding required for the union to end at a natural boundary of its strictest member.

### Declaring a Union



*identifier* Provides a tag name for the union. If specified, subsequent declarations (in the same scope) of variables using the union can be made by referring to the tag name. If not specified, you must place all variable definitions that refer to the union within the declaration of the data type.

*member* The list of members provides the data type with a description of the values that can be stored in the union.



A member that does not represent a bit field can be of any data type and can have the **volatile** or **const** qualifier.

If a `:` (colon) and a constant expression follow the member declarator, the member represents a *bit field*. Bit fields are described in Declaring and Using Bit Fields in Structures (page 97).

You can reference one of the possible members of a union the same way as referencing a member of a structure.

For example:

```
union {
    char birthday[9];
    int age;
    float weight;
} people;
people.birthday[0] = '\n';
```

assigns `'\n'` to the first element in the character array `birthday`, a member of the union `people`.

A union can represent only one of its members at a time. In the example, the union `people` contains either `age`, `birthday`, or `weight` but never more than one of these. The `printf` statement in the following example does not give the correct result because `people.age` replaces the value assigned to `people.birthday` in the first line:

```
1 people.birthday = "03/06/56";
2 people.age = 38;
3 printf("%s\n", people.birthday);
```

### Defining a Union Variable

A union variable definition contains an optional storage class keyword, the **union** keyword, a union tag, and a declarator. The union tag indicates the data type of the union variable.

The type specifier contains the keyword **union** followed by the name of the union type. You must declare the union data type before you can define a union having that type.

You can define a union data type and a union of that type in the same statement by placing the variable declarator after the data type definition.

The declarator is an identifier, possibly with the **volatile** or **const** qualifier.

The initializer must be a constant expression if the union has **static** storage duration or if you are compiling your source code in **ansi** mode. If the union has **auto** storage duration, it can be initialized using the = (equal sign) followed by any expression that returns a compatible union value. You can only initialize the first member of a union.

The following example shows how you would initialize the first union member `birthday` of the union variable `people`:

```
union {
    char birthday[9];
    int age;
    float weight;
} people = {"23/07/57"};
```

### Defining a Union Type and a Union Variable

To define union type and a union variable in one statement, put a declarator after the type definition. The storage class specifier for the variable must go at the beginning of the statement.

### Alignment of Unions

The rules for alignment of structures and structure members apply to unions, with the following exception: when the **-qalign=twobyte** option is specified, a union whose largest element is a bit field of width 16 or less has a size of 2 bytes. If the width of the bit field is greater than 16, the size of the union is 4 bytes.

### Anonymous Unions

Union can be declared without declarators if they are members of another structure or union. Unions without declarators are called *anonymous unions*.

**Note:** Anonymous unions are not part of the the ANSI C language standard, and are supported by C for AIX in **extended** compiler mode only.

Members of an anonymous union can be accessed as if they were declared directly in the containing structure or union. For example, given the following structure:

```
struct s {
    int a;
    union {
        int b;
        float c;
    };
} kurt; /* no declarator */
```

you can make the following statements:

```
kurt.a = 5;
kurt.b = 36;
```

You can also declare an anonymous union by:

1. Creating a **typedef** and using the **typedef** name without a declarator:

```
typedef union {
    int a;
    int b;
} UNION_T;
```

```

struct s1 {
    UNION_T;
    int c;
} dave;

```

1. By using an existing union tag without a declarator:

```

union u1 {
    int a;
    int b;
};
struct s1 {
    union u1;
    int c;
} dave;

```

In both of these examples, the members can be accessed as `dave.a`, `dave.b`, and `dave.c`.

An anonymous union must be a member of, or nested within another anonymous union that is a member of, a named structure or union. If a union is declared at file scope without a declarator, its members are not available to the surrounding scope. For example, the following union only declares the union tag `tom`:

```

union tom {
    int b;
    float c;
} ;

```

The variables `b` and `c` from this union cannot be used at file scope, and the following statements will generate errors:

```

b = 5;
c = 2.5;

```

### Related Reading

- “Declarators” on page 64
- “Initializers” on page 65
- “C Language Levels” on page 78
- “Example of union Declaration and Use”
- “Incomplete Types” on page 106
- “struct (Structures)” on page 95
- Structure and Union Member Specification (page 119)
- “Data Type Qualifiers” on page 115
- “static” on page 112
- “auto” on page 106
- “typedef” on page 115
- “align” on page 234

## Example of union Declaration and Use

The following example defines a union data type (not named) and a union variable (named `length`). The member of `length` can be a **long int**, a **float**, or a **double**.

```

union {
    float meters;
    double centimeters;
    long inches;
} length;

```

The following example defines the union type data as containing one member. The member can be named `charctr`, `whole`, or `real`. The second statement defines two data type variables: `input` and `output`.

```

union data {
    char charctr;
    int whole;
    float real;
};
union data input, output;

```

The following statement assigns a character to `input`:

```
input.charctr = 'h';
```

The following statement assigns a floating-point number to member `output`:

```
output.real = 9.2;
```

The following example defines an array of structures named `records`. Each element of `records` contains three members: the integer `id_num`, the integer `type_of_input`, and the union variable `input`, which has the union data type defined in the previous example.

```

struct {
    int id_num;
    int type_of_input;
    union data input;
} records[10];

```

The following statement assigns a character to the structure member `input` of the first element of `records`:

```
records[0].input.charctr = 'g';
```

#### Related Reading

“union (Unions)” on page 103

## Incomplete Types

Incomplete types are the type **void**, an array of unknown size, or structure, union, or enumeration tags that have no member lists. For example, the following are incomplete types:

```

void *incomplete_ptr;
struct dimension linear; /* no previous definition of dimension */

```

**void** is an incomplete type that cannot be completed. Incomplete structure or union and enumeration tags must be completed before being used to declare an object, although you can define a pointer to an incomplete structure or union.

#### Related Reading

“void” on page 85

“Arrays” on page 86

“struct (Structures)” on page 95

“union (Unions)” on page 103

“void” on page 85

## auto

The **auto** storage class specifier lets you define a variable with automatic storage; its use and storage is restricted to the current block. The storage class keyword **auto** is optional in a data declaration. It is not permitted in a parameter declaration. A variable having the **auto** storage class specifier must be declared within a block. It cannot be used for file scope declarations.

Because automatic variables require storage only while they are actually being used, defining variables with the **auto** storage class can decrease the amount of memory required to run a program. However, having many large automatic objects may cause you to run out of stack space.

Declaring variables with the **auto** storage class can also make code easier to maintain, because a change to an **auto** variable in one function never affects another function (unless it is passed as an argument).

The following example lines declare variables having the **auto** storage class specifier:

```
auto int counter;
auto char letter = 'k';
```

### Initialization

You can initialize any **auto** variable except parameters. If you do not initialize an automatic object, its value is indeterminate. If you provide an initial value, the expression representing the initial value can be any valid C expression. For structure and union members, the initial value must be a valid constant expression if an initializer list is used. The object is then set to that initial value each time the program block that contains the object's definition is entered.

**Note:** If you use the **goto** statement to jump into the middle of a block, automatic variables within that block are not initialized.

### Storage

Objects with the **auto** storage class specifier have automatic storage duration. Each time a block is entered, storage for **auto** objects defined in that block is made available. When the block is exited, the objects are no longer available for use.

If an **auto** object is defined within a function that is recursively invoked, memory is allocated for the object at each invocation of the block.

### Related Reading

"Block Scope Data Declarations" on page 62  
"Examples Using auto Storage Classes"  
"goto" on page 151

## Examples Using auto Storage Classes

The following program shows the scope and initialization of **auto** variables. The function `main` defines two variables, each named `auto_var`. The first definition occurs on line 10. The second definition occurs in a nested block on line 13. While the nested block is running, only the `auto_var` created by the second definition is available. During the rest of the program, only the `auto_var` created by the first definition is available.

```
1  /******
2  ** Example illustrating the use of auto variables **
3  *****/
4
5  #include <stdio.h>
6
7  int main(void)
8  {
9      void call_func(int passed_var);
10     auto int auto_var = 1; /* first definition of auto_var */
11
12     {
13         int auto_var = 2; /* second definition of auto_var */
14         printf("inner auto_var = %d\n", auto_var);
15     }
16     call_func(auto_var);
17     printf("outer auto_var = %d\n", auto_var);
```

```

18     return 0;
19 }
20
21 void call_func(int passed_var)
22 {
23     printf("passed_var = %d\n", passed_var);
24     passed_var = 3;
25     printf("passed_var = %d\n", passed_var);
26 }

```

This program produces the following output:

```

inner auto_var = 2
passed_var = 1
passed_var = 3
outer auto_var = 1

```

The following example uses an array that has the storage class **auto** to pass a character string to the function `sort`. The function `sort` receives the address of the character string, rather than the contents of the array. The address enables `sort` to change the values of the elements in the array.

```

/*****
** Sorted string program – this example passes an array name **
** to a function **
*****/
#include <stdio.h>
#include <string.h>
int main(void)
{
    void sort(char *array, int n);
    char string[75];
    int length;
    printf("Enter letters:\n");
    scanf("%74s", string);
    length = strlen(string);
    sort(string,length);
    printf("The sorted string is: %s\n", string);
    return(0);
}

void sort(char *array, int n)
{
    int gap, i, j, temp;
    for (gap = n / 2; gap > 0; gap /= 2)
        for (i = gap; i <n; i++)
            for (j=i gap; j>= 0 && array[j] > array[j + gap];
                j -= gap)
            {
                temp = array[j];
                array[j] = array[j + gap];
                array[j + gap] = temp;
            }
}

```

When the program is run, interaction with the program could produce:

```

Output      Enter letters:
Input       zyfab
Output      The sorted string is: abfyz

```

### Related Reading

“auto” on page 106



## extern

The **extern** storage class specifier lets you declare objects and functions that several source files can use. All object declarations that occur outside a function and that do not contain a storage class specifier declare identifiers with external linkage. All function definitions that do not specify a storage class define functions with external linkage.

An **extern** variable, function definition, or declaration also makes the described variable or function usable by the succeeding part of the current source file. This declaration does not replace the definition. The declaration is used to describe the variable that is externally defined.

If a declaration for an identifier already exists at file scope, any **extern** declaration of the same identifier found within a block refers to that same object. If no other declaration for the identifier exists at file scope, the identifier has external linkage.

An **extern** declaration can appear outside a function or at the beginning of a block. If the declaration describes a function or appears outside a function and describes an object with external linkage, the keyword **extern** is optional.

If you do not specify a storage class specifier, the function has external linkage.

### Initialization

You can initialize any object with the **extern** storage class specifier at file scope. You can initialize an **extern** object with an initializer that must either:

- Appear as part of the definition and the initial value must be described by a constant expression. OR
- Reduce to the address of a previously declared object with static storage duration. This object may be modified by adding or subtracting an integral constant expression.

If you do not explicitly initialize an **extern** variable, its initial value is zero of the appropriate type. Initialization of an **extern** object is completed by the time the program starts running.

### Storage

Storage is allocated at compile time for **extern** variables that are initialized. Uninitialized variables are mapped at compile time and initialized to 0 (zero) at load time. This storage is freed when the program finishes running.

### Related Reading

“Constant Expressions” on page 71

“File Scope Data Declarations” on page 63

“Examples Using extern Storage Classes”

“Function Declarations” on page 138

## Examples Using extern Storage Classes

The following program shows the linkage of **extern** objects and functions. The **extern** object `total` is declared on line 12 of `File 1` and on line 11 of `File 2`. The definition of the external object `total` appears in `File 3`. The **extern** function `tally` is defined in `File 2`. The function `tally` can be in the same file as `main` or in a different file. Because `main` precedes these definitions and `main` uses both `total` and `tally`, `main` declares `tally` on line 11 and `total` on line 12.

### File 1

```

1  /*****
2  ** The program receives the price of an item, adds the      **
3  ** tax, and prints the total cost of the item.              **
4  *****/
5  #include <stdio.h>
6
7  int main(void)
8  {
9      /* begin main */
10     void tally(void); /* declaration of function tally */
11     extern float total; /* first declaration of total */
12
13     printf("Enter the purchase amount: \n");
14     tally();
15     printf("\nWith tax, the total is: %.2f\n", total);
16
17     return(0);
18 } /* end main */

```

## File 2

```

1  /*****
2  ** This file defines the function tally                      **
3  *****/
4  #include <stdio.h>
5
6  #define tax_rate 0.05
7
8  void tally(void)
9  {
10     /* begin tally */
11     float tax;
12     extern float total; /* second declaration of total */
13
14     scanf("%f", &total);
15     tax = tax_rate * total;
16     total += tax;
17 } /* end tally */

```

## File 3

```

1  float total;

```

When this program is run, interaction with it could produce:

```

Output      Enter the purchase amount:
Input       99.95
Output      With tax, the total is: 104.95

```

The following program shows **extern** variables used by two functions. Because both functions main and sort can access and change the values of the **extern** variables string and length, main does not have to pass parameters to sort.

```

/*****
** Sorted string program – this example shows extern      **
** used by two functions                                  **
*****/
#include <stdio.h>
#include <string.h>
char string[75];

```

```

int length;
int main(void)
{
    void sort(void);
    printf("Enter letters:\n");
    scanf("%s", string);
    length = strlen(string);
    sort();
    printf("The sorted string is: %s\n", string);
    return(0);
}

void sort(void)
{
    int gap, i, j, temp;
    for (gap = length / 2; gap > 0; gap /= 2)
        for (i = gap; i <length; i++)
            for (j = i - gap;
                j >= 0 && string[j] > string[j + gap];
                j -= gap)
                {
                    temp = string[j];
                    string[j] = string[j + gap];
                    string[j + gap] = temp;
                }
}

```

When this program is run, interaction with it could produce:

Output	Enter letters:
Input	zyfab
Output	The sorted string is: abfyz

### Related Reading

“extern” on page 109

## register

The **register** storage class specifier indicates to the compiler that a heavily used variable (such as a loop control variable) within a block scope data definition or a parameter declaration should be allocated a register to minimize access time.

It is equivalent to the **auto** storage class except that the compiler places the object, if possible, into a machine register for faster access.

**Note:** Because the C for AIX compiler optimizes register use, it ignores the **register** keyword.

Most heavily-used entities are generated by the compiler itself; therefore, register variables are given no special priority for placement in machine registers. The **register** storage class keyword is required in a data definition and in a parameter declaration that describes an object having the **register** storage class. An object having the **register** storage class specifier must be defined within a block or declared as a parameter to a function.

The following example lines define automatic storage duration objects using the **register** storage class specifier:

```

register int score1 = 0, score2 = 0;
register unsigned char code = 'A';
register int *element = &order[0];

```

## Initialization

You can initialize any **register** object except parameters. If you do not initialize an automatic object, its value is indeterminate. If you provide an initial value, the expression representing the initial value can be any valid C expression. For structure and union members, the initial value must be a valid constant expression if an initializer list is used. The object is then set to that initial value each time the program block that contains the object's definition is entered.

## Storage

Objects with the **register** storage class specifier have automatic storage duration. Each time a block is entered, storage for **register** objects defined in that block are made available. When the block is exited, the objects are no longer available for use.

If a **register** object is defined within a function that is recursively invoked, the memory is allocated for the variable at each invocation of the block.

The **register** storage class specifier indicates that the object is heavily used and indicates to the compiler that the value of the object should reside in a machine register. Because of the limited size and number of registers available on most systems, few variables can actually be put in registers.

If the compiler does not allocate a machine register for a **register** object, the object is treated as having the storage class specifier **auto**.

Using register definitions for variables that are heavily used may make your object files smaller and make them run faster. In object code, a reference to a register can require less code and time than a reference to memory. In C programs, even if a **register** variable is treated as a variable with storage class **auto**, the address of the variable cannot be taken.

## Restrictions

You cannot use the **register** storage class specifier in file scope data declarations.

You cannot apply the address (&) operator to **register** variables.

## Related Reading

Block Scope Data Declarations  
"auto" on page 106

## static

The **static** storage class specifier lets you define objects with static storage duration and internal linkage, or to define functions with internal linkage.

An object having the **static** storage class specifier can be defined within a block or at file scope. If the definition occurs within a block, the object has no linkage. If the definition occurs at file scope, the object has internal linkage.

## Initialization

You can initialize any **static** object with a constant expression or an expression that reduces to the

address of a previously declared **extern** or static object, possibly modified by a constant expression. If you do not provide an initial value, the object receives the value of zero of the appropriate type.

## Storage

Storage is allocated at compile time for static variables that are initialized. Uninitialized static variables are mapped at compile time and initialized to 0 (zero) at load time. This storage is freed when the program finishes running. Beyond this, the language does not define the order of initialization of objects from different files.

## Block Scope Usage

Use **static** variables to declare objects that retain their value from one execution of a block to the next execution of that block. The **static** storage class specifier keeps the variable from being reinitialized each time the block where the variable is defined runs. For example:

```
static float rate = 10.5;
```

Initialization of a **static** array is performed only once at compile time. The following examples show the initialization of an array of characters and an array of integers:

```
static char message[] = "startup completed";
static int integers[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

## File Scope Usage

The **static** storage class specifier causes the variable to be visible only in the file where it is declared. Files, therefore, cannot access file scope **static** variables declared in other files.

## Restrictions

You cannot declare a **static** function at block scope.

## Related Reading

“Block Scope Data Declarations” on page 62

“File Scope Data Declarations” on page 63

“Examples Using static Storage Classes”

“Function Declarations” on page 138

“extern” on page 109

## Examples Using static Storage Classes

The following program shows the linkage of **static** identifiers at file scope. This program uses two different external **static** identifiers named `stat_var`. The first definition occurs in File 1. The second definition occurs in File 2. The `main` function references the object defined in File 1.. The `var_print` function references the object defined in File 2.

### File 1

```
/* *****
** Program to illustrate file scope static variables          **
***** */
#include <stdio.h>
extern void var_print(void);
static stat_var = 1;
int main(void)
```

```

{
    printf("file1 stat_var = %d\n", stat_var);
    var_print();
    printf("FILE1 stat_var = %d\n", stat_var);
    return(0);
}

```

## File 2

```

/*****
** This file contains the second definition of stat_var      **
*****/
#include <stdio.h>
static int stat_var = 2;
void var_print(void)
{
    printf("file2 stat_var = %d\n", stat_var);
}

```

This program produces the following output:

```

file1 stat_var = 1
file2 stat_var = 2
FILE1 stat_var = 1

```

The following program shows the linkage of **static** identifiers with block scope. The test function defines the static variable `stat_var`, which retains its storage throughout the program, even though `test` is the only function that can refer to `stat_var`.

```

/*****
** Program to illustrate block scope static variables      **
*****/
#include <stdio.h>
int main(void)
{
    void test(void);
    int counter;
    for (counter = 1; counter <= 4; ++counter)
        test();
    return(0);
}
void test(void)
{
    static int stat_var = 0;
    auto int auto_var = 0;
    stat_var++;
    auto_var++;
    printf("stat_var = %d auto_var = %d\n", stat_var, auto_var);
}

```

This program produces the following output:

```

stat_var = 1 auto_var = 1
stat_var = 2 auto_var = 1
stat_var = 3 auto_var = 1
stat_var = 4 auto_var = 1

```

## Related Reading

“static” on page 112

## typedef

```
▶▶ typedef —type_specifier— identifier— ; ▶▶
```

A **typedef** declaration lets you define your own identifiers that can be used in place of type specifiers such as **int**, **float**, and **double**. The names you define using **typedef** are not new data types. They are synonyms for the data types or combinations of data types they represent.

A **typedef** declaration does not reserve storage.

When an object is defined using a **typedef** identifier, the properties of the defined object are exactly the same as if the object were defined by explicitly listing the data type associated with the identifier.

The following statements declare **LENGTH** as a synonym for **int**, then use this **typedef** to declare **length**, **width**, and **height** as integral variables.

```
typedef int LENGTH;
LENGTH length, width, height;
```

The following declarations are equivalent to the above declaration:

```
int length, width, height;
```

Similarly, you can use **typedef** to define a **struct** type. For example:

```
typedef struct {
    int scruples;
    int drams;
    int grains;
} WEIGHT;
```

The structure **WEIGHT** can then be used in the following declarations:

```
WEIGHT chicken, cow, horse, whale;
```

### Related Reading

“Type Specifiers” on page 66

“struct (Structures)” on page 95

“int, long, short” on page 81

---

## Data Type Qualifiers

Qualifier	Description
<b>const</b>	Explicitly declares a data object as a data item that cannot be changed. Its value is set at initialization. You cannot use <b>const</b> data objects in expressions requiring a modifiable lvalue. For example, a <b>const</b> data object cannot appear on the left-hand side of an assignment statement.

## volatile

Maintains consistency of memory access to data objects. It tells the compiler that the variable should always contain its current value even when optimized, so that the variable can be queried when an exception occurs. Volatile objects are read from memory each time their value is needed, and written back to memory each time they are changed.

The **volatile** qualifier is useful for data objects having values that may be changed in ways unknown to your program (such as the system clock). Portions of an expression that reference **volatile** objects are not to be changed or removed.

**Note:** These type qualifiers are only meaningful in expressions that are lvalues.

For a **volatile** or **const** pointer, you must put the keyword between the \* and the identifier. For example:

```
int * volatile x;    /* x is a volatile pointer to an int */
int * const y = &z; /* y is a const pointer to the int variable z */
```

For a pointer to a **volatile** or **const** data object, the type specifier, qualifier, and storage class specifier can be in any order. For example:

```
volatile int *x1;    /* x1 is a pointer to a volatile int */
int volatile *x2;    /* x2 is a pointer to a volatile int */
const int *y1;       /* y1 is a pointer to a const int */
int const *y2;       /* y2 is a pointer to a const int */
```

In the following example, the pointer to y is a constant. You can change the value that y points to, but you cannot change the value of y:

```
int * const y
```

In the following example, the value that y points to is a constant integer and cannot be changed. However, you can change the value of y:

```
const int * y
```

For other types of **volatile** and **const** variables, the position of the keyword within the definition (or declaration) is less important. For example:

```
volatile struct omega {
    int limit;
    char code;
} group;
```

provides the same storage as:

```
struct omega {
    int limit;
    char code;
} volatile group;
```

In both examples above, only the structure variable group receives the **volatile** qualifier. Similarly, if you specified the **const** keyword instead of **volatile**, only the structure variable **group** receives the **const** qualifier. The **const** and **volatile** qualifiers when applied to a structure or union also apply to the members of the structure or union.

Although enumeration, structure, and union variables can receive the **volatile** or **const** qualifier, enumeration, structure, and union tags do not carry the **volatile** or **const** qualifier. For example, the blue structure does not carry the **volatile** qualifier:



```
volatile struct whale {
    int weight;
    char name[8];
} beluga;
struct whale blue;
```

The keywords **volatile** and **const** cannot separate the keywords **enum**, **struct**, and **union** from their tags.

You cannot declare or define a **volatile** or **const** function but you can define or declare a function that returns a pointer to a **volatile** or **const** object.

You can put more than one qualifier on a declaration but you cannot specify the same qualifier more than once on a declaration.

If you put a type definition in the same declaration as a definition of a variable having the **volatile** or **const** qualifier, the qualifier applies to that variable only. For example:

```
enum shape { round, square, triangular, oblong } volatile object;
enum shape appearance;
```

The variable `object` is defined as **volatile**. The variable `appearance` does not have the **volatile** qualifier. Similarly, if you specified the **const** keyword instead of **volatile**, only the variable `object` receives the **const** qualifier.

## Expression Operators

### Operator Precedence and Associativity Table

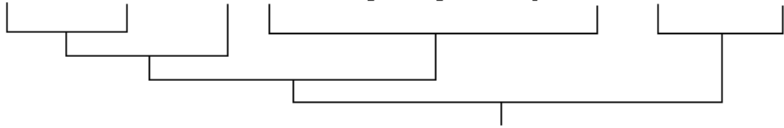

The following table lists the C language operators in order of precedence and shows the direction of associativity for each operator. The primary operators have the highest precedence. The comma operator has the lowest precedence. Operators that appear in the same group have the same precedence.

Operator Name	Associativity	Operators
Primary	left to right	() [] . ->
Unary	right to left	++ - + - ! ~ & * (type_name) sizeof
Multiplicative	left to right	* / %
Additive	left to right	+ -
Bitwise Shift	left to right	<< >>
Relational	left to right	< > <= >=
Equality	left to right	== !=
Bitwise AND	left to right	&
Bitwise Exclusive OR	left to right	^
Bitwise Inclusive OR	left to right	
Logical AND	left to right	&&
Logical OR	left to right	
Conditional	right to left	? :
Assignment	right to left	= += -= *= /= <<= >>= %= &= ^=  =
Comma	left to right	,

## Related Reading

- "Expressions and Operators" on page 67
- "Types of Expressions" on page 70
- "Primary Operators"
- "Unary Operators" on page 120
- "Binary Operators" on page 124
- "Conditional Operator (?)" on page 131
- "Assignment Operators" on page 133
- "Comma Operator ( , )" on page 136

## Primary Operators

Operators	Description
( )	<p data-bbox="302 604 805 636"><b>Parentheses Used for Expression Grouping</b></p> <p data-bbox="302 657 1398 772">Use parentheses to explicitly force the order of expression evaluation. The following expression does not contain any parentheses used for grouping operands and operators. The parentheses surrounding <code>weight, zipcode</code> are used to form a function call. Note how the compiler groups the operands and operators in the expression according to the rules for operator precedence and associativity:</p> <pre data-bbox="302 810 1081 957">-discount * item + handling(weight, zipcode) &lt; 10 * item</pre>  <p data-bbox="302 989 1398 1045">The following expression is similar to the previous expression, but it contains parentheses that change how the operands and operators are grouped:</p> <pre data-bbox="302 1083 1157 1230">{-discount * (item + handling(weight, zipcode))} &lt; {10 * item}</pre>  <p data-bbox="302 1262 1417 1346">In an expression that contains both associative and commutative operators, you can use parentheses to specify the grouping of operands with operators. The parentheses in the following expression guarantee the order of grouping operands with the operators:</p> <pre data-bbox="367 1356 558 1381">x = f + (g + h);</pre>

<p>[ ]</p>	<p><b>Array Subscripts</b></p> <p>A primary expression followed by an expression in [ ] (square brackets) specifies an element of an array. The expression within the square brackets is referred to as a <i>subscript</i>.</p> <p>The primary expression must have a pointer type, and the subscript must have integral type. The result of an array subscript is an lvalue.</p> <p>The first element of each array has the subscript 0. The expression <code>contract[35]</code> refers to the 36th element in the array <code>contract</code>.</p> <p>In a multidimensional array, you can reference each element (in the order of increasing storage locations) by incrementing the rightmost subscript most frequently.</p> <p>For example, the following statement gives the value 100 to each element in the array <code>code[4][3][6]</code>:</p> <pre> for (first = 0; first &lt;= 3; ++first)     for (second = 0; second &lt;= 2; ++second)         for (third = 0; third &lt;= 5; ++third)             code[first][second][third] = 100; </pre>
<p>.</p> <p>-&gt;</p>	<p><b>Structure and Union Member Specification</b></p> <p>Two primary operators let you specify structure and union members. The dot (a period) and arrow (formed by a minus and a greater than symbol) operators are always preceded by a primary expression and followed by an identifier.</p> <p>When you use the dot operator, the primary expression must be an instance of a type of structure or union, and the identifier must name a member of that structure or union. The result is the value associated with the named structure or union member. The result is an lvalue if the first expression is an lvalue.</p> <p>Some sample dot expressions:</p> <pre> roster[num].name roster[num].name[1] </pre> <p>When you use the arrow operator, the primary expression must be a pointer to a structure or a union, and the identifier must name a member of the structure or union. The result is the value of the named structure or union member to which the pointer expression refers. In the following example, <code>name</code> is an <b>int</b>:</p> <pre> roster -&gt; name </pre>

### Related Reading

- “Operator Precedence and Associativity” on page 67
- “Expressions and Operators” on page 67
- “Types of Expressions” on page 70
- “Operator Precedence and Associativity Table” on page 117
- “Unary Operators” on page 120
- “Binary Operators” on page 124
- “Conditional Operator (?)” on page 131
- “Assignment Operators” on page 133
- “Comma Operator ( , )” on page 136
- “Arrays” on page 86
- “struct (Structures)” on page 95
- “union (Unions)” on page 103
- “int, long, short” on page 81

## Unary Operators

Operators	Description
++	<p><b>Increment</b></p> <p>The ++ (increment) operator adds 1 to the value of a scalar operand, or if the operand is a pointer, increments the operand by the size of the object to which it points. The operand receives the result of the increment operation. The operand must be a modifiable lvalue of arithmetic or pointer type.</p> <p>You can put the ++ before or after the operand. If it appears before the operand, the operand is incremented, and then the incremented value is used in the expression. If you put the ++ after the operand, the value of the operand is used in the expression <i>before</i> the operand is incremented. For example:</p> <pre>play = ++play1 + play2++;</pre> <p>is equivalent to the following three expressions:</p> <pre>play1 = play1 + 1; play = play1 + play2; play2 = play2 + 1;</pre> <p>Because the order of evaluation for subexpressions is not specified, avoid using a variable more than once in an expression in which the variable is incremented. For example, the following expression might cause <i>i</i> to be incremented before or after the function <i>x</i> is called:</p> <pre>y = x(i) + i++;</pre> <p>The result has the same type as the operand after integral promotion, but is not an lvalue. The usual arithmetic conversions on the operand are performed.</p>
-	<p><b>Decrement</b></p> <p>The — (decrement) operator subtracts 1 from the value of a scalar operand, or if the operand is a pointer, decreases the operand by the size of the object to which it points. The operand receives the result of the decrement operation. The operand must be a modifiable lvalue.</p> <p>You can put the — before or after the operand. If it appears before the operand, the operand is decremented, and the decremented value is used in the expression. If the — appears after the operand, the current value of the operand is used in the expression and the operand is decremented.</p> <p>For example:</p> <pre>play = --play1 + play2--;</pre> <p>is equivalent to the following three expressions:</p> <pre>play1 = play1 - 1; play = play1 + play2; play2 = play2 - 1;</pre> <p>Because the order of evaluation for subexpressions is not specified, avoid using a variable more than once in an expression in which the variable is decremented. For example, the following expression might cause <i>i</i> to be decremented before or after the function <i>x</i> is called:</p> <pre>y = x(i) + i--;</pre> <p>The result has the same type as the operand after integral promotion, but is not an lvalue. The usual arithmetic conversions on the operand are performed.</p>

+	<p><b>Unary Plus</b></p> <p>The + (unary plus) operator maintains the value of the operand. The operand can have any arithmetic type. The result is not an lvalue.</p> <p>The result of the unary plus expression has the same type as the operand after any integral promotions (for example, <b>char</b> to <b>int</b>). The usual arithmetic conversions on the operand are performed.</p> <p><b>Note:</b> Any plus sign in front of a constant is not part of the constant.</p>
-	<p><b>Unary Minus</b></p> <p>The + (unary plus) operator negates the value of the operand. The operand can have any arithmetic type. The result is not an lvalue.</p> <p>For example, if <code>quality</code> has the value 100, <code>-quality</code> has the value -100.</p> <p>The result of the unary minus expression has the same type as the operand after any integral promotions (for example, <b>char</b> to <b>int</b>). The usual arithmetic conversions on the operand are performed.</p> <p><b>Note:</b> Any plus sign in front of a constant is not part of the constant.</p>
!	<p><b>Logical Negation</b></p> <p>The ! (logical negation) operator determines whether the operand evaluates to 0 (false) or nonzero (true). The expression yields the value 1 (true) if the operand evaluates to 0, and the value 0 (false) if the operand evaluates to a nonzero value. The operand must have a scalar data type, but the result of the operation has always type <b>int</b> and is not an lvalue.</p> <p>The following two expressions are equivalent:</p> <pre>!right; right == 0;</pre> <p>The usual arithmetic conversions on the operand are performed.</p>
~	<p><b>Bitwise Negation</b></p> <p>The ~ (bitwise negation) operator yields the bitwise complement of the operand. In the binary representation of the result, every bit has the opposite value of the same bit in the binary representation of the operand. The operand must have an integral type. The result has the same type as the operand but is not an lvalue.</p> <p>Suppose <code>x</code> represents the decimal value 5. The 32-bit binary representation of <code>x</code> is:</p> <pre>00000000000000000000000000000101</pre> <p>The expression <code>~x</code> yields the following result, represented here as a 32-bit binary number:</p> <pre>11111111111111111111111111111010</pre> <p>The 32-bit binary representation of <code>~0</code> is:</p> <pre>11111111111111111111111111111111</pre>

<p style="text-align: center;"><b>&amp;</b></p>	<p><b>Address</b></p> <p>The &amp; (address) operator yields a pointer to its operand. The operand must be an lvalue or function designator. It cannot be a bit field, nor can it have the storage class <b>register</b>.</p> <p>If the operand is an lvalue or function, the resulting type is a pointer to the expression type. For example, if the expression has type <b>int</b>, the result is a pointer to an object having type <b>int</b>. The result is not an lvalue.</p> <p>If <code>p_to_y</code> is defined as a pointer to an <b>int</b> and <code>y</code> as an <b>int</b>, the following expression assigns the address of the variable <code>y</code> to the pointer <code>p_to_y</code>:</p> <pre style="margin-left: 40px;">p_to_y = &amp;y;</pre>
<p style="text-align: center;"><b>*</b></p>	<p><b>Indirection</b></p> <p>The * (indirection) operator determines the value referred to by the pointer-type operand.</p> <p>The operand cannot be a pointer to an incomplete type. The operation yields an lvalue or a function designator if the operand points to a function. Arrays and functions are converted to pointers.</p> <p>The type of the operand determines the type of the result. For example, if the operand is a pointer to an <b>int</b>, the result has type <b>int</b>.</p> <p>Do not apply the indirection operator to any pointer that contains an address that is not valid, such as <b>NULL</b>. The result is not defined.</p> <p>If <code>p_to_y</code> is defined as a pointer to an <b>int</b> and <code>y</code> as an <b>int</b>, the expressions:</p> <pre style="margin-left: 40px;">p_to_y = &amp;y; *p_to_y = 3;</pre> <p>cause the variable <code>y</code> to receive the value 3.</p>
<p><b>Cast</b>(<i>type_name</i>)</p>	<p><b>Type Casting</b></p> <p>A <i>cast</i> operator converts the type of the operand to a specified data type and performs the necessary conversions to the operand for the type.</p> <p>The cast operator is a type specifier in parentheses. This type and the operand must be scalar. The type can also be <b>void</b>. The result has the type of the specified data type but is not an lvalue.</p> <p>The following expression contains a cast expression, <code>(double)x</code>, to convert an operand of type <b>int</b> to a value of type <b>double</b>:</p> <pre style="margin-left: 40px;">int x; printf("x=%lf\n", (double)x);</pre> <p>The function <code>printf</code> receives the value of <code>x</code> as a <b>double</b>. The variable <code>x</code> remains unchanged by the cast.</p>

<b>sizeof</b>	<p><b>Size of an Object</b></p> <p>The <b>sizeof</b> operator yields the size in <i>bytes</i> of the operand. Types cannot be defined in a <b>sizeof</b> expression. Except in <b>extended</b> mode C, the <b>sizeof</b> operation cannot be performed on</p> <ul style="list-style-type: none"> <li>• A bit field</li> <li>• A function</li> <li>• An array with unspecified dimensions</li> <li>• An incomplete type (such as <b>void</b>)</li> </ul> <p>The operand can be the parenthesized name of a type or expression.</p> <p>The compiler must be able to evaluate the size at compile time. The expression is not evaluated; there are no side effects. For example, the value of <i>b</i> is 5 from initialization to the end of program runtime:</p> <pre style="margin-left: 2em;">#include &lt;stdio.h&gt; int main(void){     int b = 5;     sizeof(b++); }</pre> <p>When the compiler is in 32-bit mode, the result is an integer constant. When the compiler is in 64-bit mode, the result is an unsigned long.</p> <p>The size of a <b>char</b> object is the size of a byte. For example, if a variable <i>x</i> has type <b>char</b>, the expression <code>sizeof(x)</code> always evaluates to 1.</p> <p>The result of a <code>sizeof</code> operation has type <b>size_t</b>, which is an unsigned integral type defined in the <b>&lt;stddef.h&gt;</b> header. Header files are described in the <i>AIX Version 4 Files Reference</i>.</p> <p>The size of an object is determined on the basis of its definition. The <b>sizeof</b> operator does not perform any conversions. If the operand contains operators that perform conversions, the compiler does take these conversions into consideration. The following expression causes the usual arithmetic conversions to be performed. The result of the expression <code>x + 1</code> has type <b>int</b> (if <i>x</i> has type <b>char</b>, <b>short</b>, or <b>int</b> or any enumeration type) and is equivalent to <code>sizeof(int)</code>:</p> <pre style="margin-left: 2em;">sizeof (x + 1);</pre> <p>Except in preprocessor directives, you can use a <b>sizeof</b> expression wherever an integral constant is required. One of the most common uses for the <b>sizeof</b> operator is to determine the size of objects that are referred to during storage allocation, input, and output functions.</p> <p>Another use of <b>sizeof</b> is in porting code across platforms. You should use the <b>sizeof</b> operator to determine the size that a data type represents. For example:</p> <pre style="margin-left: 2em;">sizeof(int);</pre> <p>Using the <b>sizeof</b> operator on a bit field is not permitted in <b>ansi</b> mode. It is allowed in <b>extended</b> mode, and returns the same result as <code>sizeof(int)</code>.</p> <p>When applied to a C enumeration constant, <b>sizeof</b> always returns 4 because enumeration constants in C always have type <b>int</b> or <b>unsigned int</b>. When applied to an enumeration compiled under the <b>-qenum=small</b> option, the result of the <b>sizeof</b> operation is the size of the predefined type used to allocate storage for the enumeration.</p>
---------------	---

## Related Reading

“Operator Precedence and Associativity” on page 67  
“Expressions and Operators” on page 67  
“Types of Expressions” on page 70  
“Operator Precedence and Associativity Table” on page 117  
“Primary Operators” on page 118  
“Binary Operators”  
“Conditional Operator (?)” on page 131  
“Assignment Operators” on page 133  
“Comma Operator ( , )” on page 136  
“Arithmetic Conversions Table” on page 137  
“Pointers” on page 90  
“register” on page 111  
“char” on page 79  
“int, long, short” on page 81  
“float, double” on page 80  
“void” on page 85  
“enum” on page 255

## Binary Operators

Operators	Description
*	<p><b>Multiplication</b></p> <p>The * (multiplication) operator yields the product of its operands. The operands must have an arithmetic type. The result is not an lvalue. The usual arithmetic conversions on the operands are performed.</p> <p>Because the multiplication operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one multiplication operator. For example, the expression:</p> <pre>sites * number * cost</pre> <p>can be interpreted in any of the following ways:</p> <pre>(sites * number) * cost sites * (number * cost) (cost * sites) * number</pre>
/	<p><b>Division</b></p> <p>The / (division) operator yields the quotient of its operands. The operands must have an arithmetic type. The result is not an lvalue.</p> <p>If both operands are positive integers and the operation produces a remainder, the remainder is ignored. For example, expression <code>7 / 4</code> yields the value 1 (rather than 1.75 or 2).</p> <p>On all IBM C compilers, if either operand is negative, the result is rounded towards zero.</p> <p>The result is undefined if the second operand evaluates to 0.</p> <p>For more information on generating warning messages for division by constant zero, see the <b>-qinfo</b> compiler option.</p> <p>The usual arithmetic conversions on the operands are performed.</p>



%	<p><b>Remainder</b></p> <p>The % (remainder) operator yields the remainder from the division of the left operand by the right operand. For example, the expression <code>5 % 3</code> yields 2. The result is not an lvalue.</p> <p>Both operands must have an integral type. If the right operand evaluates to 0, the result is undefined. If either operand has a negative value, the result is such that the following expression always yields the value of a if b is not 0 and <code>a / b</code> is representable:</p> <pre>( a / b ) * b + a % b;</pre> <p>The sign of the remainder is the same as the sign of the quotient.</p> <p>The usual arithmetic conversions on the operands are performed.</p>
+	<p><b>Addition</b></p> <p>The + (addition) operator yields the sum of its operands. Both operands must have an arithmetic type, or one operand must be a pointer to an object type and the other operand must have an integral type.</p> <p>When both operands have an arithmetic type, the usual arithmetic conversions on the operands are performed. The result has the type produced by the conversions on the operands and is not an lvalue.</p> <p>A pointer to an object in an array can be added to a value having integral type. The result is a pointer of the same type as the pointer operand. The result refers to another element in the array, offset from the original element by the amount specified by the integral value. If the resulting pointer points to storage outside the array, other than the first location outside the array, the result is undefined. The compiler does not provide boundary checking on the pointers. For example, after the addition, <code>ptr</code> points to the third element of the array:</p> <pre>int array[5]; int *ptr; ptr = array + 2;</pre>
-	<p><b>Subtraction</b></p> <p>The - (subtraction) operator yields the difference of its operands. Both operands must have an arithmetic type, or the left operand must have a pointer type and the right operand must have the same pointer type or an integral type. You cannot subtract a pointer from an integral value.</p> <p>When both operands have an arithmetic type, the usual arithmetic conversions on the operands are performed. The result has the type produced by the conversions on the operands and is not an lvalue.</p> <p>When the left operand is a pointer and the right operand has an integral type, the compiler converts the value of the right to an address offset. The result is a pointer of the same type as the pointer operand.</p> <p>If both operands are pointers to the same type, the compiler converts the result to an integral type that represents the number of objects separating the two addresses. Behavior is undefined if the pointers do not refer to objects in the same array.</p>

<<  
>>

## Bitwise Shifts

The bitwise shift operators move the bit values of a binary object. The left operand specifies the value to be shifted. The right operand specifies the number of positions that the bits in the value are to be shifted. The result is not an lvalue. Both operands have the same precedence and are left-to-right associative.

### Operator

Usage

<< Indicates the bits are to be shifted to the left.

>> Indicates the bits are to be shifted to the right.

Each operand must have an integral type. The compiler performs integral promotions on the operands. Then the right operand is converted to type **int**. The result has the same type as the left operand (after the arithmetic conversions).

The right operand should not have a negative value or a value that is greater than or equal to the width in bits of the expression being shifted. The result of bitwise shifts on such values is unpredictable.

If the right operand has the value 0, the result is the value of the left operand (after the usual arithmetic conversions).

The << operator fills vacated bits with zeros. For example, if `left_op` has the value 4019, the bit pattern (in 32-bit format) of `left_op` is:

```
000000000000000000000000111110110011
```

The expression `left_op << 3` yields:

```
000000000000000000000000111110110011000
```

The following table shows the behavior of the >> operator:

### Left Operand Type

Result of >>

#### unsigned type

The vacated bits are filled with zeros.

#### Nonnegative unsigned type

The integral part of the quotient of the left operand divided by the quantity 2, raised to the power of the right operand. The vacated bits of a signed value are filled with a copy of the sign bit of the unshifted value.

#### Negative signed type

The language does not specify how the vacated bits produced by the >> operator are filled.

<p>&lt; &gt; &lt;= &gt;=</p>	<p><b>Relational</b></p> <p>The relational operators compare two operands and determine the validity of a relationship. If the relationship stated by the operator is true, the value of the result is 1. If false, the value of the result is 0. The result is not an lvalue.</p> <p>The following table describes the four relational operators:</p> <table border="1"> <thead> <tr> <th>Operator</th> <th>Usage</th> </tr> </thead> <tbody> <tr> <td>&lt;</td> <td>Indicates whether the value of the left operand is less than the value of the right operand.</td> </tr> <tr> <td>&gt;</td> <td>Indicates whether the value of the left operand is greater than the value of the right operand.</td> </tr> <tr> <td>&lt;=</td> <td>Indicates whether the value of the left operand is less than or equal to the value of the right operand.</td> </tr> <tr> <td>&gt;=</td> <td>Indicates whether the value of the left operand is greater than or equal to the value of the right operand.</td> </tr> </tbody> </table> <p>Both operands must have arithmetic types or be pointers to the same type. The result has type <b>int</b>.</p> <p>If the operands have arithmetic types, the usual arithmetic conversions on the operands are performed.</p> <p>When the operands are pointers, the result is determined by the locations of the objects to which the pointers refer. If the pointers do not refer to objects in the same array, the result is not defined.</p> <p>Relational operators have left-to-right associativity. For example, the expression:</p> $a < b <= c$ <p>is interpreted as:</p> $(a < b) <= c$ <p>If the value of a is less than the value of b, the first relationship is true and yields the value 1. The compiler then compares the value 1 with the value of c.</p>	Operator	Usage	<	Indicates whether the value of the left operand is less than the value of the right operand.	>	Indicates whether the value of the left operand is greater than the value of the right operand.	<=	Indicates whether the value of the left operand is less than or equal to the value of the right operand.	>=	Indicates whether the value of the left operand is greater than or equal to the value of the right operand.
Operator	Usage										
<	Indicates whether the value of the left operand is less than the value of the right operand.										
>	Indicates whether the value of the left operand is greater than the value of the right operand.										
<=	Indicates whether the value of the left operand is less than or equal to the value of the right operand.										
>=	Indicates whether the value of the left operand is greater than or equal to the value of the right operand.										

==  
!=

## Equality

The equality operators, like the relational operators, compare two operands for the validity of a relationship. The equality operators, however, have a lower precedence than the relational operators. If the relationship stated by an equality operator is true, the value of the result is 1. Otherwise, the value of the result is 0.

The following table describes the two equality operators:

### Operator

#### Usage

**==** Indicates whether the value of the left operand is equal to the value of the right operand.

**!=** Indicates whether the value of the left operand is not equal to the value of the right operand.

Both operands must have arithmetic types or be pointers to the same type, or one operand must have a pointer type and the other operand must be a pointer to void or NULL. The result has type **int**.

If the operands have arithmetic types, the usual arithmetic conversions on the operands are performed.

If the operands are pointers, the result is determined by the locations of the objects to which the pointers refer.

If one operand is a pointer and the other operand is an integer having the value 0, the **==** expression is true only if the pointer operand evaluates to NULL. The **!=** operator evaluates to true if the pointer operand does *not* evaluate to NULL.

You can also use the equality operators to compare pointers to members that are of the same type but do not belong to the same object. The following expressions contain examples of equality and relational operators:

```
time < max_time == status < complete  
letter != EOF
```

**Note:** The equality operator (**==**) should not be confused with the assignment (**=**) operator.

For example,

```
if(x == 3)
```

evaluates to 1 if x is equal to three. Equality tests like this should be coded with spaces between the operator and the operands to prevent unintentional assignments.

```
if(x = 3)
```

is taken to be true because  $(x = 3)$  evaluates to a non-zero value (3). The expression also assigns the value 3 to x.

<p><b>&amp;</b></p>	<p><b>Bitwise AND</b></p> <p>The &amp; (bitwise AND) operator compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1's, the corresponding bit of the result is set to 1. Otherwise, it sets the corresponding result bit to 0.</p> <p>Both operands must have an integral type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands.</p> <p>Because the bitwise AND operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one bitwise AND operator.</p> <p>The following example shows the values of a, b, and the result of a &amp; b represented as 32-bit binary numbers:</p> <p><b>bit pattern of a</b>  000000000000000000000000000000001011100</p> <p><b>bit pattern of b</b>  00000000000000000000000000000000101110</p> <p><b>bit pattern of a &amp; b</b>  00000000000000000000000000000000100</p> <p><b>Note:</b> The bitwise AND (&amp;) should not be confused with the logical AND (&amp;&amp;) operator. For example, 1 &amp; 4 evaluates to 0 while 1 &amp;&amp; 4 evaluates to 1</p>
<p><b>^</b></p>	<p><b>Bitwise Exclusive OR</b></p> <p>The bitwise exclusive OR operator compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1's or both bits are 0's, the corresponding bit of the result is set to 0. Otherwise, it sets the corresponding result bit to 1.</p> <p>Both operands must have an integral type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands and is not an lvalue.</p> <p>Because the bitwise exclusive OR operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one bitwise exclusive OR operator even when the sub-expressions are explicitly grouped with parentheses.</p> <p>The following example shows the values of a, b, and the result of a ^ b represented as 32-bit binary numbers:</p> <p><b>bit pattern of a</b>  000000000000000000000000000000001011100</p> <p><b>bit pattern of b</b>  00000000000000000000000000000000101110</p> <p><b>bit pattern of a ^ b</b>  00000000000000000000000000000000110010</p>

	<p><b>Bitwise Inclusive OR</b></p> <p>The   (bitwise inclusive OR) operator compares the values (in binary format) of each operand and yields a value whose bit pattern shows which bits in either of the operands has the value 1. If both of the bits are 0, the result of that bit is 0; otherwise, the result is 1.</p> <p>Both operands must have an integral type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands and is not an lvalue.</p> <p>Because the bitwise inclusive OR operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one bitwise inclusive OR operator even when the subexpressions are explicitly grouped with parentheses.</p> <p>The following example shows the values of a, b, and the result of a   b represented as 32-bit binary numbers:</p> <p><b>bit pattern of a</b> 0000000000000000000000001011100</p> <p><b>bit pattern of b</b> 000000000000000000000000101110</p> <p><b>bit pattern of a   b</b> 000000000000000000000000111110</p> <p><b>Note:</b> The bitwise OR ( ) should not be confused with the logical OR (  ) operator. For example, 1   4 evaluates to 5 while 1    4 evaluates to 1</p>								
&&	<p><b>Logical AND</b></p> <p>The &amp;&amp; (logical AND) operator indicates whether both operands have a nonzero value. If both operands have nonzero values, the result has the value 1. Otherwise, the result has the value 0.</p> <p>Both operands must have a scalar type. The usual arithmetic conversions on each operand are performed. The result has type <b>int</b> and is not an lvalue.</p> <p>Unlike the &amp; (bitwise AND) operator, the &amp;&amp; operator guarantees left-to-right evaluation of the operands. If the left operand evaluates to 0, the right operand is not evaluated.</p> <p>The following examples show how the expressions that contain the logical AND operator are evaluated:</p> <table border="0"><thead><tr><th data-bbox="303 1255 435 1281"><b>Expression</b></th><th data-bbox="399 1283 469 1308">Result</th></tr></thead><tbody><tr><td data-bbox="303 1329 412 1354">1 &amp;&amp; 0</td><td data-bbox="399 1329 412 1354">0</td></tr><tr><td data-bbox="303 1373 412 1398">1 &amp;&amp; 4</td><td data-bbox="399 1373 412 1398">1</td></tr><tr><td data-bbox="303 1417 412 1442">0 &amp;&amp; 0</td><td data-bbox="399 1417 412 1442">0</td></tr></tbody></table> <p>The following example uses the logical AND operator to avoid division by zero:</p> <pre data-bbox="370 1507 592 1533">(y != 0) &amp;&amp; (x / y)</pre> <p>The expression x / y is not evaluated when y != 0 evaluates to 0.</p> <p><b>Note:</b> The logical AND (&amp;&amp;) should not be confused with the bitwise AND (&amp;) operator. For example: 1 &amp;&amp; 4 evaluates to 1 while 1 &amp; 4 evaluates to 0</p>	<b>Expression</b>	Result	1 && 0	0	1 && 4	1	0 && 0	0
<b>Expression</b>	Result								
1 && 0	0								
1 && 4	1								
0 && 0	0								

<p><b>  </b></p>	<p><b>Logical OR</b></p> <p>The    (logical OR) operator indicates whether either operand has a nonzero value. If either operand has a nonzero value, the result has the value 1. Otherwise, the result has the value 0.</p> <p>Both operands must have a scalar type. The usual arithmetic conversions on each operand are performed. The result has type <b>int</b> and is not an lvalue.</p> <p>Unlike the   (bitwise inclusive OR) operator, the    operator guarantees left-to-right evaluation of the operands. If the left operand has a nonzero value, the right operand is not evaluated.</p> <p>The following examples show how expressions that contain the logical OR operator are evaluated:</p> <table style="border-collapse: collapse; margin-top: 10px;"> <thead> <tr> <th style="text-align: left; padding-right: 10px;"><b>Expression</b></th> <th style="text-align: left;"><b>Result</b></th> </tr> </thead> <tbody> <tr> <td style="padding-top: 5px;">1    0</td> <td style="padding-top: 5px;">1</td> </tr> <tr> <td style="padding-top: 5px;">1    4</td> <td style="padding-top: 5px;">1</td> </tr> <tr> <td style="padding-top: 5px;">0    0</td> <td style="padding-top: 5px;">0</td> </tr> </tbody> </table> <p>The following example uses the logical OR operator to conditionally increment y:</p> <pre style="margin-left: 20px;">++x    ++y;</pre> <p>The expression ++y is not evaluated when the expression ++x evaluates to a nonzero quantity.</p> <p><b>Note:</b> The logical OR (    ) should not be confused with the bitwise OR (   ) operator. For example: 1    4 evaluates to 1 while 1   4 evaluates to 5</p>	<b>Expression</b>	<b>Result</b>	1    0	1	1    4	1	0    0	0
<b>Expression</b>	<b>Result</b>								
1    0	1								
1    4	1								
0    0	0								

### Related Reading

- “Operator Precedence and Associativity” on page 67
- “Expressions and Operators” on page 67
- “Types of Expressions” on page 70
- “Arithmetic Conversions” on page 76
- “Standard Type Conversions” on page 74
- “Pointer Conversions” on page 75
- “Operator Precedence and Associativity Table” on page 117
- “Primary Operators” on page 118
- “Unary Operators” on page 120
- “Conditional Operator (?)”
- “Assignment Operators” on page 133
- “Comma Operator ( , )” on page 136
- “Arithmetic Conversions Table” on page 137
- Pointer Arithmetic (page 92)
- “int, long, short” on page 81
- “info” on page 275

## Conditional Operator (?)

A *conditional expression* is a compound expression that contains a condition (*operand1*), an expression to be evaluated if the condition has a non-zero value (*operand2*), and an expression to be evaluated if the condition has the value 0 (*operand3*).

Conditional expressions have right-to-left associativity. The left operand (*operand1*) is evaluated first, and then only one of the two remaining operands is evaluated. If that operand’s expression contains or returns arithmetic types, the usual arithmetic conversions are performed on that expression’s values.

The conditional expression contains one two-part operator. The ? symbol follows the condition, and the : appears between the two action expressions. All expressions that occur between the ? and : are treated as one expression.

The first operand must have a scalar type. The type of the second and third operands must be one of the following:

- an arithmetic type
- a compatible pointer, structure, or union type
- **void**

The second and third operands can also be a pointer or a null pointer constant.

Two object are compatible when they have the same type, but not necessarily the same type qualifiers (**volatile** or **const**). Pointer objects are compatible if they have the same type, or are pointers to **void**.

The first expression is evaluated first. If the first expression returns a non-zero value, the second expression is evaluated, converted to the result type, and becomes the value of the conditional expression. The third operand is ignored in this case. If the first expression instead returns a zero value, the third operand is evaluated, converted to the result type, and becomes the value of the conditional expression. The second expression is ignored in this case.

The types of the second and third operands determine the type of the result as shown below:

Type of One Operand	Type of Other Operand	Type of Result
Arithmetic	Arithmetic	Arithmetic type after usual arithmetic conversions
Structure or union type	Compatible structure or union type	Structure or union type with all the qualifiers on both operands
void	void	void
Pointer to compatible type	Pointer to compatible type	Pointer to type with all the qualifiers specified for the type
Pointer to type	NULL pointer (the constant 0)	Pointer to type
Pointer to object or incomplete type	Pointer to void	Pointer to void with all the qualifiers specified for the type

### Related Reading

- “Operator Precedence and Associativity” on page 67
- “Expressions and Operators” on page 67
- “Types of Expressions” on page 70
- “Examples Using the Conditional Operator”
- “Operator Precedence and Associativity Table” on page 117
- “Primary Operators” on page 118
- “Unary Operators” on page 120
- “Binary Operators” on page 124
- “Assignment Operators” on page 133
- “Comma Operator ( , )” on page 136
- “Arithmetic Conversions Table” on page 137
- Pointer Arithmetic (page 92)
- “void” on page 85

### Examples Using the Conditional Operator

The following expression determines which variable has the greater value, y or z, and assigns the greater value to the variable x.

```
x = (y > z) ? y : z;
```

The following is an equivalent statement:



```
if (y > z)
    x = y;
else
    x = z;
```

The following expression calls the function `printf`, which receives the value of the variable `c` if `c` evaluates to a digit. Otherwise, `printf` receives the character constant `'x'`.

```
printf(" c = %c\n", isdigit(c) ? c : 'x');
```

If the last operand of a conditional expression contains an assignment operator, use parentheses to ensure the expression evaluates properly. For example, the `==` operator has higher precedence than the `?:` operator in the following expression:

```
int i, j, k;
(i == 7) ? j ++ : k = j;
```

This expression generates an error because it is interpreted as if it were parenthesized this way:

```
int i, j, k;
((i == 7) ? j ++ : k) = j;
```

The value `k`, and not `k = j`, is treated as the third operand. This error arises because a conditional expression is not an lvalue, and the assignment is not valid. To make the expression evaluate correctly, enclose the last operand in parentheses. For example:

```
int i, j, k;
(i == 7) ? j ++ : (k = j);
```

### Related Reading

“Operator Precedence and Associativity” on page 67

“Expressions and Operators” on page 67

“Types of Expressions” on page 70

“lvalues” on page 70

“Operator Precedence and Associativity Table” on page 117

“Conditional Operator (?)” on page 131

## Assignment Operators

Operators	Description
-----------	-------------

=

### Simple Assignment

The simple assignment operator stores the value of the right operand in the object designated by the left operand.

Both operands must have arithmetic types, the same structure type, or the same union type. Otherwise, both operands must be pointers to the same type, or the left operand must be a pointer and the right operand must be the constant 0 or NULL. If the language level is **extended**, both operands can be pointers to different types.

If both operands have arithmetic types, the system converts the type of the right operand to the type of the left operand before the assignment.

If the left operand is a pointer and the right operand is the constant 0, the result is NULL.

Pointers to void can appear on either side of the simple assignment operator.

A packed structure or union can be assigned to a nonpacked structure or union of the same type, and a nonpacked structure or union can be assigned to a packed structure or union of the same type.

If one operand is packed and the other is not, the layout of the right operand is remapped to match the layout of the left. This remapping of structures might degrade performance. For efficiency, when you perform assignment operations with structures or unions, you should ensure that both operands are either packed or nonpacked.

**Note:** If you assign pointers to structures or unions, the objects they point to must both be either packed or nonpacked.

You can assign values to operands with the type qualifier **volatile**. You cannot assign a pointer of an object with the type qualifier **const** to a pointer of an object without the **const** type qualifier. For example:

```
const int *p1;
int *p2;
p2 = p1; /* this is NOT allowed */
p1 = p2; /* this IS allowed */
```

**Note:** The assignment (=) operator should not be confused with the equality operator (==).

For example,

```
if(x == 3)
```

evaluates to 1 if x is equal to three. Equality tests like this should be coded with spaces between the operator and the operands to prevent unintentional assignments.

```
if(x = 3)
```

is taken to be true because (x = 3) evaluates to a non-zero value (3). The expression also assigns the value 3 to x.

+=	<p><b>Compound Assignment</b></p> <p>The compound assignment operators consist of a binary operator and the simple assignment operator. They perform the operation of the binary operator on both operands and give the result of that operation to the left operand.</p> <p>The following table shows the operand types of compound assignment expressions:</p> <table border="1"> <thead> <tr> <th>Operator</th> <th>Left Operand</th> <th>Right Operand</th> </tr> </thead> <tbody> <tr> <td>+=or -=</td> <td>Arithmetic</td> <td>Arithmetic</td> </tr> <tr> <td>+=or -=</td> <td>Pointer</td> <td>Integral type</td> </tr> <tr> <td>*, /*, and %/</td> <td>Arithmetic</td> <td>Arithmetic</td> </tr> <tr> <td>&lt;&lt;=, &gt;&gt;=, &amp;=, ^=, and  =</td> <td>Integral type</td> <td>Integral type</td> </tr> </tbody> </table> <p>Note that the expression <math>a *= b + c</math> is equivalent to <math>a = a * (b + c)</math>, and <i>not</i> <math>a = a * b + c</math>.</p>	Operator	Left Operand	Right Operand	+=or -=	Arithmetic	Arithmetic	+=or -=	Pointer	Integral type	*, /*, and %/	Arithmetic	Arithmetic	<<=, >>=, &=, ^=, and  =	Integral type	Integral type
Operator		Left Operand	Right Operand													
+=or -=		Arithmetic	Arithmetic													
+=or -=		Pointer	Integral type													
*, /*, and %/		Arithmetic	Arithmetic													
<<=, >>=, &=, ^=, and  =		Integral type	Integral type													
-=																
*=																
/=																
%=																
<<=																
>>=																
&=																
^=																
=																

### Related Reading

- “Operator Precedence and Associativity” on page 67
- “Expressions and Operators” on page 67
- “Types of Expressions” on page 70
- “Examples Using Compound Assignment Operators”
- “Operator Precedence and Associativity Table” on page 117
- “Primary Operators” on page 118
- “Unary Operators” on page 120
- “Binary Operators” on page 124
- “Conditional Operator (?)” on page 131
- “Comma Operator ( , )” on page 136
- “Data Type Qualifiers” on page 115

### Examples Using Compound Assignment Operators

The table below lists the compound assignment operators and shows an expression using each operator:

Operator	Example	Equivalent Expression
+=	index += 2	index = index + 2
-=	*(pointer++) -= 1	*pointer = *(pointer++) - 1
*=	bonus *= increase	bonus = bonus * increase
/=	time /= hours	time = time / hours
%=	allowance %= 1000	allowance = allowance % 1000
<<=	result <<= num	result = result << num
>>=	form >>= 1	form = form >> 1
&=	mask &= 2	mask = mask & 2

<code>^=</code>	<code>test ^= pre_test</code>	<code>test = test ^ pre_test</code>
<code> =</code>	<code>flag  = ON</code>	<code>flag = flag   ON</code>

Although the equivalent expression column shows the left operands (from the example column) evaluated twice, the left operand is evaluated only once.

### Related Reading

“Assignment Operators” on page 133

## Comma Operator ( , )

A *comma expression* contains two operands separated by a comma. Although the compiler evaluates both operands, the value of the right operand is the value of the expression. The left operand is evaluated, possibly producing side effects, and the value is discarded. The result of a comma expression is not an lvalue.

Both operands of a comma expression can have any type. All comma expressions have left-to-right associativity. The left operand is fully evaluated before the right operand.

In the following example, if `omega` has the value 11, the expression increments `delta` and assigns the value 3 to `alpha`:

```
alpha = (delta++, omega % 4);
```

Any number of expressions separated by commas can form a single expression. The compiler evaluates the leftmost expression first. The value of the rightmost expression becomes the value of the entire expression.

For example, the value of the expression:

```
intensity++, shade * increment, rotate(direction);
```

is the value of the expression:

```
rotate(direction)
```

The primary use of the comma operator is to produce side effects in the following situations:

- Calling a function
- Entering or repeating an iteration loop
- Testing a condition
- Other situations where a side effect is required but the result of the expression is not immediately needed

To use the comma operator in a context where the comma has other meanings, such as in a list of function arguments or a list of initializers, you must enclose the comma operator in parentheses. For example, the function

```
f(a, (t = 3, t + 2), c);
```

has only three arguments: the value of `a`, the value 5, and the value of `c`. The value of the second argument is the result of the comma expression in parentheses:

```
t = 3, t + 2
```

which has the value 5.

### Related Reading

- “Operator Precedence and Associativity” on page 67
- “Expressions and Operators” on page 67
- “Types of Expressions” on page 70
- “Examples Using the Comma Operator ( , )”
- “Operator Precedence and Associativity Table” on page 117
- “Primary Operators” on page 118
- “Unary Operators” on page 120
- “Binary Operators” on page 124
- “Conditional Operator (?)” on page 131
- “Assignment Operators” on page 133

## Examples Using the Comma Operator ( , )

The table below gives some examples of the uses of the comma operator:

Statement	Effects
<code>for (i=0; i&lt;2; ++i, f() );</code>	A <b>for</b> statement in which <code>i</code> is incremented and <code>f()</code> is called at each iteration.
<code>if ( f(), ++i, i&gt;1 ) { /* ... */ }</code>	An <b>if</b> statement in which function <code>f()</code> is called, variable <code>i</code> is incremented, and variable <code>i</code> is tested against a value. The first two expressions within this comma expression are evaluated before the expression <code>i&gt;1</code> . Regardless of the results of the first two expressions, the third is evaluated and its result determines whether the <b>if</b> statement is processed.
<code>func( ( ++a, f(a) ) );</code>	A function call to <code>func()</code> in which <code>a</code> is incremented, the resulting value is passed to a function <code>f()</code> , and the return value of <code>f()</code> is passed to <code>func()</code> . The function <code>func()</code> is passed only a single argument, because the comma expression is enclosed in parentheses within the function argument list.

### Related Reading

- “Comma Operator ( , )” on page 136

---

## Arithmetic Conversions Table

Arithmetic conversions are used for matching operands of arithmetic operators, and proceed in the following order:

Operand Type	Conversion
One operand has <b>long double</b> type	The other operand is converted to <b>long double</b> type.
One operand has <b>double</b> type	The other operand is converted to <b>double</b> .
One operand has <b>float</b> type	The other operand is converted to <b>float</b> .
One operand has <b>unsigned long int</b> type	The other operand is converted to <b>unsigned long int</b> .
One operand has <b>unsigned long long int</b> type	The other operand is converted to <b>unsigned long long int</b> .
One operand has <b>unsigned int</b> type and the other operand has <b>long int</b> type and the value of the <b>unsigned int</b> can be represented in a <b>long int</b>	The operand with <b>unsigned int</b> type is converted to <b>long int</b> .
One operand has <b>unsigned int</b> type and the other operand has <b>long int</b> type and the value of the <b>unsigned int</b> cannot be represented in a <b>long int</b>	Both operands are converted to <b>unsigned long int</b>
One operand has <b>long int</b> type	The other operand is converted to <b>long int</b> .

Operand Type	Conversion
One operand has <b>long long int</b> type	The other operand is converted to <b>long long int</b> .
One operand has <b>unsigned int</b> type	The other operand is converted to <b>unsigned int</b> .
Both operands have <b>int</b> type	The result is type <b>int</b> .

**Note:** The rules for arithmetic conversions in **extended** mode are different, and are described in “Arithmetic Conversions for extended Level C” on page 167.

#### Related Reading

“Arithmetic Conversions” on page 76

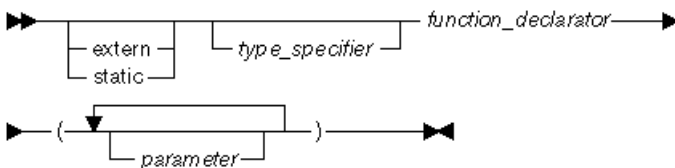
“Arithmetic Conversions for extended Level C” on page 167

---

## Functions

### Function Declarations

A function declaration establishes the name and the parameters of the function.



A function is declared implicitly by its appearance in an expression if it has not been defined or declared previously; the implicit declaration is equivalent to a declaration of `extern int func_name()`. The default return type of a function is “**int, long, short**” on page 81.

To indicate that the function does not return a value, declare it with a return type of “**void**” on page 85.

A function cannot be declared as returning a data object having a “**Data Type Qualifiers**” on page 115 or “**Data Type Qualifiers**” on page 115 type but it can return a pointer to a **volatile** or **const** object. Also, a function cannot return a value that has a type of array or function.

If the called function returns a value that has a type other than “**int, long, short**” on page 81, you must declare the function before the function call. Even if a called function returns a type **int**, explicitly declaring the function prior to its call is good programming practice.

Some declarations do not have parameter lists; the declarations simply specify the types of parameters and the return values, such as in the following example:

```
int func(int, long);
```

#### Related Reading

“Functions” on page 77

“Function Calls” on page 72

“Example of the main() Function” on page 145

“Examples of Function Declarations” on page 139

“Examples of Function Definitions” on page 142

“main() Function” on page 144

“Function Definitions” on page 139

## Examples of Function Declarations

The following example defines the function `absolute` with the return type **double**. Because this is a non-integer return type, `absolute` is declared prior to the function call.

```
#include <stdio.h>
double absolute(double);
int main(void)
{
    double f = -3.0;
    printf("absolute number = %lf\n", absolute(f));
}
double absolute(double number)
{
    if (number < 0.0)
        number = -number;
    return number;
}
```

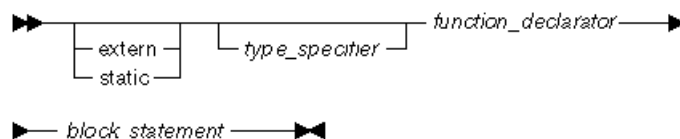
The following example defines the function `absolute` with the return type **void**. Within the function `main`, `absolute` is declared with the return type **void**.

```
#include <stdio.h>
int main(void)
{
    void absolute(float);
    float f = -8.7;
    absolute(f);
}
void absolute(float number)
{
    if (number < 0.0)
        number = -number;
    printf("absolute number = %f\n", number);
}
```

### Related Reading

- “Functions” on page 77
- “Function Calls” on page 72
- “Examples of Function Calls” on page 73
- “Example of the `main()` Function” on page 145
- “Examples of Function Definitions” on page 142
- “`main()` Function” on page 144
- “Function Declarations” on page 138
- “Function Definitions”
- “float, double” on page 80
- “void” on page 85

## Function Definitions



A function definition (either prototype or nonprototype) contains the following:

- An optional *storage class specifier* **extern** or **static**, which determines the scope of the function. If a storage class specifier is not given, the function has external linkage.
- An optional *type specifier*, which determines the type of value that the function returns. If a type specifier is not given, the function has type **int**.

- A *function declarator*, which provides the function with a name, can further describe the type of the value that the function returns, and can list any parameters that the function expects and their types. The parameters that the function is expecting are enclosed in parentheses.
- A *block statement*, which contains data definitions and code.

A nonprototype function definition can also have a list of *parameter declarations*, which describe the types of parameters that the function receives. In nonprototype functions, parameters that are not declared have type **int**.

A function can be called by itself or by other functions. Unless a function definition has the storage class specifier **static**, the function also can be called by functions that appear in other files or modules. Functions with a storage class specifier of **static** can only be directly invoked from within the same source file.

If a function has the storage class specifier **static** or a return type other than **int**, the function definition or a declaration for the function must appear before, and in the same file as, a call to the function. If a function definition has external linkage and a return type of **int**, calls to the function can be made before it is visible because an implicit declaration of `extern int func();` is assumed.

All declarations for a given function must be compatible; that is, the return type is the same and the parameters have the same type.

The default type for the return value and parameters of a function is **int**, and the default storage class specifier is **extern**. If the function does not return a value or it is not passed any parameters, use the keyword **void** as the type specifier.

You can include ellipses (...) at the end of your parameter list to indicate that a variable number of arguments will be passed to the function. Parameter promotions are performed, and no type checking is done.

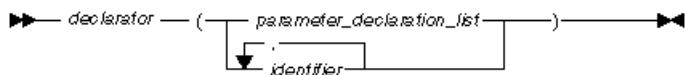
You cannot declare a function as a structure or union member.

A function cannot have a return type of function, array, or any type having the **volatile** or **const** qualifier. However, it can return a pointer to an object with a **volatile** or **const** type.

You cannot define an array of functions. You can, however, define an array of pointers to functions.

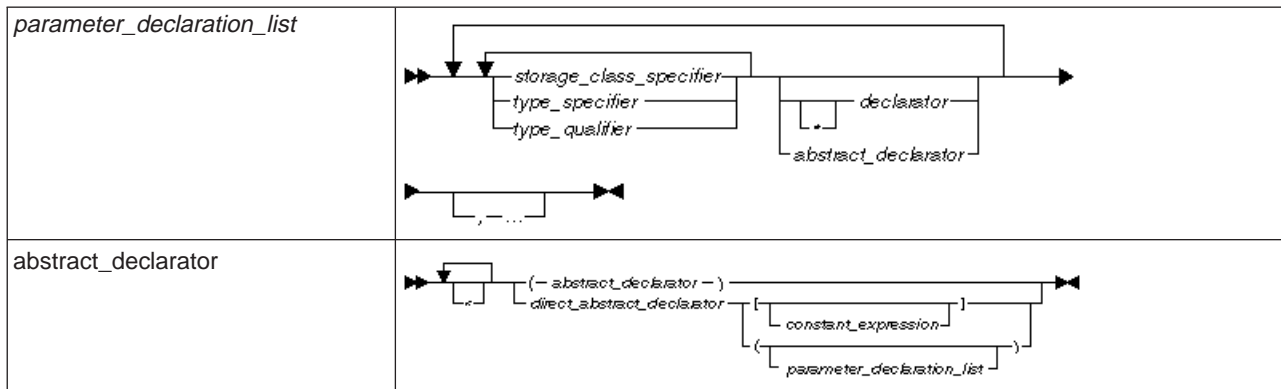
## Function Declarator

The *function declarator* shown in the function definition syntax diagram names the function and lists the function parameters. It contains an identifier that names the function and a list of the function parameters. You should always use prototype function declarators because of the parameter checking that can be performed. The detailed syntax structure for the function declarator is:





where:



### Prototype Function Declarators

Each parameter should be declared within the function declarator. Any calls to the function must pass the same number of arguments as there are parameters in the declaration.

### Nonprototype Function Declarators

Each parameter should be declared in a *parameter declaration list* following the declarator. If a parameter is not declared, it has type **int**.

**char** and **short** parameters are widened to **int**, and **float** to **double**. No type checking between the argument type and the parameter type is done for nonprototyped functions. As well, there are no checks to ensure that the number of arguments matches the number of parameters.

Each value that a function receives should be declared in a parameter declaration list for nonprototype function definitions that follows the declarator.

A parameter declaration determines the storage class specifier and the data type of the value.

The only storage class specifier allowed is the **register** storage class specifier. Any type specifier for a parameter is allowed. If you do not specify the **register** storage class specifier, the parameter will have the **auto** storage class specifier. If you omit the type specifier and you are not using the prototype form to define the function, the parameter will have type **int**.

```
int func(i,j)
{
    /* i and j have type int */
}
```

You cannot declare a parameter in the parameter declaration list if it is not listed within the declarator.

### Ellipsis and void

An ellipsis at the end of a parameter declaration indicates that the number of arguments is equal to, or greater than, the number of specified argument types. At least one parameter declaration must come before the ellipsis. Where it is permitted, an ellipsis preceded by a comma is equal to a simple ellipsis.

```
int f(int,...);
```

The comma before the ellipsis is optional.

Parameter promotions are performed as needed, but no type checking is done on the variable arguments.

You can declare a function with no arguments in two ways:

```
int f(void);    /* f() takes no parameters */
int f();       /* f() takes any number */
               /* or type or parameters */
```

An empty argument declaration list or the argument declaration list of (void) indicates a function that takes no arguments. **void** cannot be used as an argument type, although types derived from **void** (such as pointers to **void**) can be used.

In the following example, the function f() takes one integer parameter and returns no value, while g() expects no parameters and returns an integer.

```
void f(int)
int g(void)
```

## Function Body

The body of a function is a block statement.

The following function body contains a definition for the integer variable big\_num, an if-else control statement, and a call to the function **printf**:

```
void largest(int num1, int num2)
{
    int big_num;
    if (num1 >= num2)
        big_num = num1;
    else
        big_num = num2;
    printf("big_num = %d\n", big_num);
}
```

## Related Reading

- “Functions” on page 77
- “Function Calls” on page 72
- “Example of the main() Function” on page 145
- “Examples of Function Declarations” on page 139
- “Examples of Function Definitions”
- “main() Function” on page 144
- “Function Declarations” on page 138
- “extern” on page 109
- “register” on page 111
- “static” on page 112
- “char” on page 79
- “float, double” on page 80
- “int, long, short” on page 81
- “void” on page 85

## Examples of Function Definitions

In the following example, ary is an array of two function pointers. Type casting is performed to the values assigned to ary for compatibility:

```
#include <stdio.h>
int func1(void);
void func2(double a);
int main(void)
{
```

```

double num;
int retnum;
void (*ary[2]) ();
ary[0] = ((void(*)())func1);
ary[1] = ((void(*)())func2);
((int (*)())ary[0])();          /* calls func1 */
((void (*)(double))ary[1])(num); /* calls func2 */
}
int func1(void)
{
    int number=3;
    return number;
}
void func2(double a)
{
    a=333.3333;
}

```

The following example is a complete definition of the function sum:

```

int sum(int x,int y)
{
    return(x + y);
}

```

The function sum has external linkage, returns an object that has type **int**, and has two parameters of type **int** declared as x and y. The function body contains a single statement that returns the sum of x and y.

The following example contains a function declarator sort with table declared as a pointer to **int**, and length declared as type **int**. Note that arrays as parameters are implicitly converted to a pointer to the type.

```

void sort(int table[], int length)
{
    int i, j, temp;
    for (i = 0; i < length - 1; i++)
        for (j = i + 1; j < length; j++)
            if (table[i] > table[j])
                {
                    temp = table[i];
                    table[i] = table[j];
                    table[j] = temp;
                }
}

```

The following examples contain prototype function declarators:

```

double square(float x);
int area(int x, int y);
static char *search(char);

```

The following example shows how a **typedef** function can be used in a function declarator:

```

typedef struct tm_fmt { int minutes;
                      int hours;
                      char am_pm;
                      } struct_t;
long time_seconds(struct_t arrival)

```

The following function set\_date declares a pointer to a structure of type date as a parameter. date\_ptr has the storage class specifier **register**.

```

set_date(register struct date *date_ptr)
{
    date_ptr->mon = 12;
    date_ptr->day = 25;
    date_ptr->year = 87;
}

```

### Related Reading

- “Functions” on page 77
- “Function Calls” on page 72
- “Examples of Function Calls” on page 73
- “Example of the main() Function” on page 145
- “Examples of Function Declarations” on page 139
- “main() Function”
- “Function Declarations” on page 138
- “Function Definitions” on page 139
- “typedef” on page 115
- “register” on page 111
- “int, long, short” on page 81

## main() Function

The function **main** can be declared with or without arguments that pass program parameters and environment settings to the program. Although any name can be given to these parameters, they are usually referred to as *argc*, *argv*, and *envp*.

<i>argc</i>	Is the argument count. It has type <b>int</b> and indicates how many arguments are entered on the command line.
<i>argv</i>	Is the argument vector. It is an array of pointers to <b>char</b> array objects. These <b>char</b> objects are null-terminated strings that are the program arguments passed to the program when it is invoked.
<i>envp</i>	Is an optional environment pointer. It is an array of pointers to <b>char</b> objects that are the environment variables available to the program. These have the form <i>name=value</i> . The system determines the value of this parameter during program initialization (before calling <b>main</b> ). Because you can use the function <b>getenv</b> to get the value of these pointers, there is usually no need to declare this parameter.

The value of *argc* indicates the number of pointers in the array *argv*. If a program name is available, the first element in *argv* points to a character array that contains the program name or the invocation name of the program that is being run. If the name cannot be determined, the first element in *argv* points to a null character.

This name is counted as one of the arguments to the function **main**. For example, if only the program name is entered on the command line, *argc* has a value of 1 and *argv[0]* points to the program name.

Regardless of the number of arguments entered on the command line, *argv[argc]* always contains NULL.

### Related Reading

- “Functions” on page 77
- “Function Calls” on page 72
- “Statement Blocks” on page 159
- “Type Specifiers” on page 66
- “Examples of Function Calls” on page 73
- “Examples of Function Declarations” on page 139
- “Examples of Function Definitions” on page 142
- “Function Declarations” on page 138
- “Statement Blocks” on page 159
- “char” on page 79
- “int, long, short” on page 81

## Example of the main() Function

The following program backward prints the arguments entered on a command line such that the last argument is printed first:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    while (--argc > 0)
        printf("%s ", argv[argc]);
}
```

Invoking this program from a command line with the following:

```
backward string1 string2
```

gives the following output:

```
string2 string1
```

The arguments *argc* and *argv* would contain the following values:

Object	Value
argc	3
argv[0]	pointer to string"backward"
argv[1]	pointer to string"string1"
argv[2]	pointer to string"string2"
argv[3]	NULL

### Related Reading

"Functions" on page 77

"Function Calls" on page 72

"Examples of Function Declarations" on page 139

"Examples of Function Definitions" on page 142

"main() Function" on page 144

"Function Declarations" on page 138

"Function Definitions" on page 139

---

## Program Statement Keywords

### break

A *break statement* lets you end an *iterative* (**do**, **for**, **while**) or **switch** statement and exit from it at any point other than the logical end.

▶▶ break ; ▶▶

In an iterative statement, the **break** statement ends the loop and moves control to the next statement outside the loop. Within nested statements, the **break** statement ends only the smallest enclosing **do**, **for**, **switch**, or **while** statement.

In a **switch** body, the **break** passes control out of the **switch** body to the next statement outside the **switch** body.

## Restrictions

A **break** statement can appear only in the body of an iterative statement or a **switch** statement.

## Related Reading

“Examples Using the break Statement”

“do” on page 148

“for” on page 149

“switch” on page 155

“while” on page 158

## Examples Using the break Statement

The following example shows a **break** statement in the action part of a **for** statement. If the *i*th element of the array `string` is equal to `'\0'`, the **break** statement causes the **for** statement to end.

```
for (i = 0; i < 5; i++)
{
    if (string[i] == '\0')
        break;
    length++;
}
```

The following is an equivalent **for** statement, if `string` does not contain any embedded null characters:

```
for (i = 0; (i < 5)&& (string[i] != '\0'); i++)
{
    length++;
}
```

The following example shows a **break** statement in a nested iterative statement. The outer loop goes through an array of pointers to strings. The inner loop examines each character of the string. When the **break** statement is processed, the inner loop ends and control returns to the outer loop.

```
/**
** This program counts the characters in the strings that are
** part of an array of pointers to characters. The count stops
** when one of the digits 0 through 9 is encountered
** and resumes at the beginning of the next string.
**/
#include <stdio.h>
#define SIZE 3
int main(void)
{
    static char *strings[SIZE] = { "ab", "c5d", "e5" };
    int i;
    int letter_count = 0;
    char *pointer;
    for (i = 0; i < SIZE; i++) /* for each string */
        /* for each character */
        for (pointer=strings[i]; *pointer != '\0' ; ++pointer)
        {
            /* if a number */
            if (*pointer >='0' && *pointer <= '9' )
                break;
            letter_count++;
        }
    printf("letter count=%d\n", letter_count);
}
```

The program produces the following output:

```
letter count = 4
```

The following example is a **switch** statement that contains several **break** statements. Each **break** statement indicates the end of a specific clause and ends the **switch** statement.

```

#include <stdio.h>
enum {morning, afternoon, evening} timeofday = morning;
int main(void) {
    switch (timeofday) {
        case (morning):
            printf("Good Morning\n");
            break;
        case (evening):
            printf("Good Evening\n");
            break;
        default:
            printf("Good Day, eh\n");
    }
}

```

### Related Reading

“break” on page 145

## continue

A *continue statement* lets you end the current iteration of a loop. Program control is passed from the **continue** statement to the end of the loop body.

▶▶ — continue — ; — ◀◀

The **continue** statement ends the processing of the action part of an iterative (**do**, **for**, or **while**) statement and moves control to the condition part of the statement. If the iterative statement is a **for** statement, control moves to the third expression in the condition part of the statement, then to the second expression (the test) in the condition part of the statement.

Within nested statements, the **continue** statement ends only the current iteration of the **do**, **for**, or **while** statement immediately enclosing it.

### Restrictions

A **continue** statement can only appear within the body of an iterative statement.

### Related Reading

“Examples Using the continue Statement”

“do” on page 148

“for” on page 149

“while” on page 158

## Examples Using the continue Statement

The following example shows a **continue** statement in a **for** statement. The **continue** statement causes processing to skip over those elements of the array rates that have values less than or equal to 1.

```

/**
 ** This example shows a continue statement in a for statement.
 **/
#include <stdio.h>
#define SIZE 5
int main(void)
{
    int i;
    static float rates[SIZE] = { 1.45, 0.05, 1.88, 2.00, 0.75 };
    printf("Rates over 1.00\n");
}

```

```

for (i = 0; i < SIZE; i++)
{
    if (rates[i] <= 1.00) /* skip rates <= 1.00 */
        continue;
    printf("rate = %.2f\n", rates[i]);
}
return(0);}

```

The program produces the following output:

```

Rates over 1.00
rate = 1.45
rate = 1.88
rate = 2.00

```

The following example shows a **continue** statement in a nested loop. When the inner loop encounters a number in the array strings, that iteration of the loop ends. Processing continues with the third expression of the inner loop. The inner loop ends when the '\0' escape sequence is encountered.

```

/**
 ** This program counts the characters in strings that are part
 ** of an array of pointers to characters. The count excludes
 ** the digits 0 through 9.
 **/
#include <stdio.h>
#define SIZE 3
int main(void)
{
    static char *strings[SIZE] = { "ab", "c5d", "e5" };
    int i;
    int letter_count = 0;
    char *pointer;
    for (i = 0; i < SIZE; i++) /* for each string */
        /* for each each character */
        for (pointer="strings[i];" *pointer != "\0" ; ++pointer)
            /* if a number */
            if (*pointer >= '0' && *pointer <= '9')
                continue;
            letter_count++;
    }
    printf("letter count=%d\n", letter_count);
}

```

The program produces the following output:

```

letter count = 5

```

Compare this program with the third program in “Examples Using the break Statement” on page 146, which uses the **break** statement to perform a similar function.

### Related Reading

“continue” on page 147

## do

A *do statement* repeatedly runs a statement until the test expression evaluates to **0**. Because of the order of processing, the statement is run at least once.

```

▶▶ do—statement—while—(—expression—)—;—▶▶

```



The body of the loop is run before the controlling **while** clause is evaluated. Further processing of the **do** statement depends on the value of the **while** clause. If the **while** clause does not evaluate to **0**, the statement runs again. When the **while** clause evaluates to **0**, the statement ends. The controlling expression must be evaluate to a scalar type.

A **break**, **return**, or **goto** statement can cause the processing of a **do** statement to end, even when the **while** clause does not evaluate to **0**.

#### Related Reading

“Example Using the do Statement”

“break” on page 145

“goto” on page 151

“return” on page 154

“while” on page 158

### Example Using the do Statement

The following statement prompts the user to enter a 1. If the user enters a 1, the statement ends. If not, it displays another prompt.

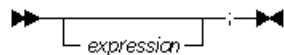
```
#include <stdio.h>
int main (void)
{
    int reply1;
    do
    {
        printf("Enter a 1\n");
        scanf("%d", &reply1);
    } while (reply1 != 1);
    return(0);
}
```

#### Related Reading

“do” on page 148

## Expressions

An *expression statement* contains an expression. The expression can be null.



An expression statement evaluates the given expression, which can then be assigned to a variable or used as an argument in a function call. Some examples are:

```
printf("Account Number: \n");      /* call to the printf */
marks = dollars * exch_rate;      /* assignment to marks */
(difference <0) ? ++losses : ++gain; /* conditional increment */
switches=flags | BIT_MASK;        /* assignment to switches */
```

#### Related Reading

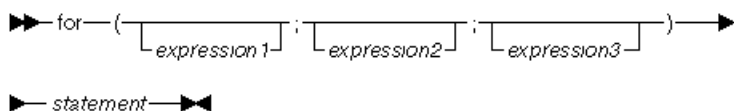
“Types of Expressions” on page 70

## for

A *for statement* lets you do the following:

- Evaluate an expression before the first iteration of the statement (*initialization*)

- Specify an expression to determine whether or not the statement should be processed (*controlling part*)
- Evaluate an expression after each iteration of the statement
- Repeatedly process the statement if the controlling part does not evaluate to zero.



Arguments to the **for** statement are:

<i>expression1</i>	Is the <i>initialization expression</i> . It is evaluated only before the <i>statement</i> is processed for the first time. You can use this expression to initialize a variable. If you do not want to evaluate an expression prior to the first iteration of the statement, you can omit this expression.
<i>expression2</i>	Is the <i>controlling part</i> . It is evaluated before each iteration of the <i>statement</i> . It must evaluate to a scalar type.  If it evaluates to <b>0</b> (zero), the statement is not processed and control moves to the next statement following the <b>for</b> statement. If <i>expression2</i> does not evaluate to <b>0</b> , the statement is processed. If you omit <i>expression2</i> , it is as if the expression had been replaced by a nonzero constant, and the <b>for</b> statement is not terminated by failure of this condition.
<i>expression3</i>	Is evaluated after each iteration of the <i>statement</i> . You can use this expression to increase, decrease, or reinitialize a variable. This expression is optional.

A **break**, **return**, or **goto** statement can cause a **for** statement to end, even when the second expression does not evaluate to **0**. If you omit *expression2*, you must use a **break**, **return**, or **goto** statement to end the **for** statement.

### Related Reading

- “Examples Using the for Statement”
- “break” on page 145
- “goto” on page 151
- “return” on page 154

## Examples Using the for Statement

The following **for** statement prints the value of `count` 20 times. The **for** statement initially sets the value of `count` to 1. After each iteration of the statement, `count` is incremented.

```
for (count = 1; count <= 20; count++)
    printf("count = %d\n", count);
```

The following sequence of statements accomplishes the same task. Note the use of the **while** statement instead of the **for** statement.

```
count = 1;
while (count <= 20)
{
    printf("count = %d\n", count);
    count++;
}
```

The following **for** statement does not contain an initialization expression:

```
for (; index > 10; -index)
{
    list[index] = var1 + var2;
    printf("list[%d] = %d\n", index, list[index]);
}
```

The following **for** statement will continue running until `scanf` receives the letter `e`.

```

for (;;)
{
    scanf("%c", &letter);
    if (letter == '\n')
        continue;
    if (letter == 'e')
        break;
    printf("You entered the letter %c\n", letter);
}

```

The following **for** statement contains multiple initializations and increments. The comma operator makes this construction possible.

```

for (i = 0, j = 50; i < 10; ++i, j += 50)
{
    printf("i = %2d and j = %3d\n", i, j);
}

```

The following example shows a nested **for** statement. It prints the values of an array having the dimensions [5][3].

```

for (row = 0; row < 5; row++)
    for (column=0; column < 3; column++)
        printf("%d\n", table[row][column]);

```

The outer statement is processed as long as the value of `row` is less than 5. Each time the outer **for** statement is executed, the inner **for** statement sets the initial value of `column` to zero and the statement of the inner **for** statement is executed 3 times. The inner statement is executed as long as the value of `column` is less than 3.

#### Related Reading

“for” on page 149

“break” on page 145

“continue” on page 147

## goto

A *goto statement* causes your program to unconditionally transfer control to the statement associated with the label specified on the **goto** statement.

```

▶▶ — goto — label_identifier — ; — ◀◀

```

Because the **goto** statement can interfere with the normal sequence of processing, it makes a program more difficult to read and maintain. Often, a **break** statement, a **continue** statement, or a function call can eliminate the need for a **goto** statement.

If you use a **goto** statement to transfer control to a statement inside of a loop or block, initializations of automatic storage for the loop do not take place and the result is undefined. The label must appear in the same function as the **goto**.

#### Related Reading

“Example Using the goto Statement” on page 152

“break” on page 145

“continue” on page 147

## Example Using the goto Statement

The following example shows a **goto** statement that is used to jump out of a nested loop. This function could be written without using a **goto** statement.

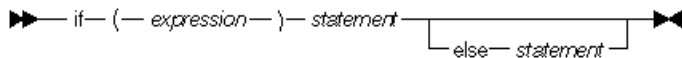
```
void display(int matrix[3][3])
{
    int i, j;
    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
            {
                if ( (matrix[i][j] < 1) || (matrix[i][j] > 6) )
                    goto out_of_bounds;
                printf("matrix[%d][%d] = %d\n", i, j, matrix[i][j]);
            }
    return;
out_of_bounds: printf("number must be 1 through 6\n");
}
```

### Related Reading

"goto" on page 151

## if / else

An *if statement* lets you conditionally process a statement when the specified test expression evaluates to a nonzero value. The expression must evaluate to a scalar type. You can optionally specify an **else** clause on the **if** statement. If the test expression evaluates to **0** and an **else** clause exists, the statement associated with the **else** clause runs. If the test expression evaluates to a nonzero value, the statement following the expression runs and the **else** clause is ignored.



When **if** statements are nested and **else** clauses are present, a given **else** is associated with the closest preceding **if** statement within the same block.

### Related Reading

"Examples Using the if/else Statement"

## Examples Using the if/else Statement

The following example causes grade to receive the value A if the value of score is greater than or equal to 90.

```
if (score >= 90)
    grade = 'A';
```

The following example displays Number is positive if the value of number is greater than or equal to 0. If the value of number is less than 0, it displays Number is negative.

```
if (number >= 0)
    printf("Number is positive\n");
else
    printf("Number is negative\n");
```

The following example shows a nested **if** statement:

```
if (paygrade == 7)
    if (level >= 0 && level <= 8)
        salary *= 1.05;
```

```

else
    salary *= 1.04;
else
    salary *= 1.06;

```

The following example shows a nested **if** statement that does not have an **else** clause. Because an **else** clause always associates with the closest **if** statement, braces might be needed to force a particular **else** clause to associate with the correct **if** statement. In this example, omitting the braces would cause the **else** clause to associate with the nested **if** statement.

```

if (kegs > 0) {
    if (furlongs > kegs)
        fpk = furlongs/kegs;
}
else
    fpk = 0;

```

The following example shows an **if** statement nested within an **else** clause. This example tests multiple conditions. The tests are made in order of their appearance. If one test evaluates to a nonzero value, a statement runs and the entire **if** statement ends.

```

if (value > 0)
    ++increase;
else if (value == 0)
    ++break_even;
else
    ++decrease;

```

#### Related Reading

“if / else” on page 152

## Null Statement

The *null statement* performs no operation.

```

▶▶—————;————▶▶

```

A null statement can hold the label of a labeled statement or complete the syntax of an iterative statement.

#### Related Reading

“Examples Using the Null Statement”

## Examples Using the Null Statement

The following example initializes the elements of the array `price`. Because the initializations occur within the **for** expressions, a statement is only needed to finish the **for** syntax; no operations are required.

```

for (i = 0; i < 3; price[i++] = 0)
    ;

```

A null statement can be used when a label is needed before the end of a block statement. For example:

```

void func(void) {
    if (error_detected)
        goto depart;
    /* further processing */
depart: ; /* null statement required */
}

```

## Related Reading

“Null Statement” on page 153

## return

A *return statement* ends the processing of the current function and returns control to the caller of the function.

```
▶▶ return expression ; ▶▶
```

A **return** statement in a function is optional. The compiler issues a warning if a return statement is not found in a function declared with a return type. If the end of a function is reached without encountering a **return** statement, control is passed to the caller as if a **return** statement without an expression were encountered. A function can contain multiple **return** statements.

### Value of a return Expression and Function Value

If an expression is present on a **return** statement, the value of the expression is returned to the caller. If the data type of the expression is different from the function return type, conversion of the return value takes place as if the value of the expression were assigned to an object with the same function return type.

If an expression is not present on a **return** statement, the value of the **return** statement is undefined. If an expression is not given on a **return** statement in a function declared with a nonvoid return type, a warning message is issued, and the result of calling the function is unpredictable.

You cannot use a **return** statement with an expression when the function is declared as returning type **void**.

## Related Reading

“Examples Using the return Statement”

“Expressions” on page 149

## Examples Using the return Statement

The following are examples of return statements:

```
return;           /* Returns no value          */
return result;    /* Returns the value of result */
return 1;         /* Returns the value 1         */
return (x * x);   /* Returns the value of x * x  */
```

The following function searches through an array of integers to determine if a match exists for the variable number. If a match exists, the function `match` returns the value of `i`. If a match does not exist, the function `match` returns the value `-1` (negative one).

```
int match(int number, int array[ ], int n)
{
    int i;
    for (i = 0; i < n; i++)
        if (number == array[i])
            return (i);
    return(-1);
}
```

## Related Reading

“return” on page 154

## switch

A *switch statement* lets you transfer control to different statements within the **switch** body depending on the value of the switch expression. The **switch** expression must evaluate to an integral value.

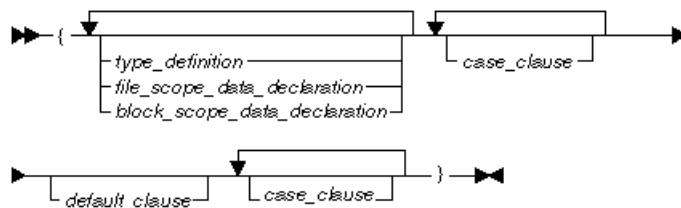
The body of the **switch** statement contains *case clauses* that consist of

- A **case** label
- An optional **default** label
- A **case** expression
- A list of statements.

If the value of the **switch** expression equals the value of one of the case expressions, the statements following that case expression are processed. If not, the default label statements, if any, are processed.

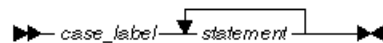
▶▶ switch ( expression ) switch\_body ▶▶

The *switch body* is enclosed in braces and can contain definitions, declarations, *case clauses*, and a *default clause*. Each case clause and default clause can contain statements.

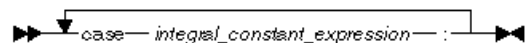


**Note:** An initializer within a *type\_definition*, *file\_scope\_data\_declaration* or *block\_scope\_data\_declaration* is ignored.

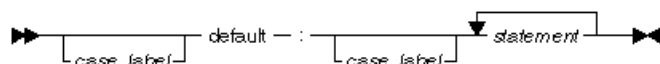
A *case clause* contains a *case label* followed by any number of statements.



A *case label* contains the word **case** followed by an integral constant expression and a colon. Anywhere you can put one **case** label, you can put multiple **case** labels.



A *default clause* contains a **default** label followed by one or more statements. You can put a **case** label on either side of the **default** label. A **switch** statement can have only one **default** label.



The **switch** statement passes control to the statement following one of the labels or to the statement following the **switch** body. The value of the expression that precedes the **switch** body determines which statement receives control. This expression is called the *switch expression*.

The value of the **switch** expression is compared with the value of the expression in each **case** label. If a matching value is found, control is passed to the statement following the **case** label that contains the matching value. If there is no matching value but there is a **default** label in the **switch** body, control passes to the **default** labelled statement. If no matching value is found, and there is no **default** label anywhere in the **switch** body, no part of the **switch** body is processed.

When control passes to a statement in the **switch** body, control only leaves the **switch** body when a **break** statement is encountered or the last statement in the **switch** body is processed.

If necessary, an integral promotion is performed on the controlling expression, and all expressions in the **case** statements are converted to the same type as the controlling expression.

### Restrictions

The **switch** expression and the **case** expressions must have an integral type. The value of each **case** expression must represent a different value and must be a constant expression.

Only one **default** label can occur in each **switch** statement. You cannot have duplicate **case** labels in a **switch** statement.

You can put data definitions at the beginning of the **switch** body, but the compiler does not initialize “**auto**” on page 106 and “**register**” on page 111 variables at the beginning of a **switch** body.

### Related Reading

“Examples Using the switch Statement”  
“break” on page 145

## Examples Using the switch Statement

The following “**switch**” on page 155 statement contains several **case** clauses and one **default** clause. Each clause contains a function call and a “**break**” on page 145 statement. The **break** statements prevent control from passing down through each statement in the **switch** body.

If the **switch** expression evaluates to `'/'`, the switch statement calls the function `divide`. Control then passes to the statement following the **switch** body.

```
char key;
printf("Enter an arithmetic operator\n");
scanf("%c",&key);
switch (key)
{
    case '+':
        add();
        break;
    case '-':
        subtract();
        break;
    case '*':
        multiply();
        break;
    case '/':
        divide();
        break;
```



```

    default:
        printf("invalid key\n");
        break;
}

```

If the switch expression matches a case expression, the statements following the case expression are processed until a **“break” on page 145** statement is encountered or the end of the **“switch” on page 155** body is reached.

In the following example, **“break” on page 145** statements are not present. If the value of `text[i]` is equal to 'A', all three counters are incremented. If the value of `text[i]` is equal to 'a', `lettera` and `total` are increased. Only `total` is increased if `text[i]` is not equal to 'A' or 'a'.

```

char text[100];
int capa, lettera, total;
for (i=0; i<sizeof(text); i++) {
    switch (text[i])
    {
        case 'A':
            capa++;
        case 'a':
            lettera++;
        default:
            total++;
    }
}

```

The following **“switch” on page 155** statement performs the same statements for more than one **case** label:

```

int month;
/* Determine what season a month falls into */
switch (month)
{
    case 12:
    case 1:
    case 2:
        printf("month %d is a winter month\n", month);
        break;
    case 3:
    case 4:
    case 5:
        printf("month %d is a spring month\n", month);
        break;
    case 6:
    case 7:
    case 8:
        printf("month %d is a summer month\n", month);
        break;
    case 9:
    case 10:
    case 11:
        printf("month %d is a fall month\n", month);
        break;
    case 66:
    case 99:
    default:
        printf("month %d is not a valid month\n", month);
}

```

If the expression `month` has the value 3, control passes to the statement:

```

printf("month %d is a spring month\n", month);

```

The **“break” on page 145** statement passes control to the statement following the **“switch” on page 155** body.

### Related Reading

“break” on page 145  
“switch” on page 155

## while

A *while statement* repeatedly runs the body of a loop until the controlling expression evaluates to **0**.

▶ while ( *expression* ) *statement* ▶

The expression is evaluated to determine whether or not to process the body of the loop. The expression must be convertible to a scalar type.

If the expression evaluates to **0**, the body of the loop never runs.

If the expression does not evaluate to **0**, the loop body is processed. After the body has run, control passes back to the expression. Further processing depends on the value of the condition.

A **break**, **return**, or **goto** statement can cause a **while** statement to end, even when the condition does not evaluate to **0**.

### Related Reading

“Example Using the While Statement”  
“break” on page 145  
“goto” on page 151  
“return” on page 154

## Example Using the While Statement

In the following program, `item[index]` triples each time the value of the expression `++index` is less than `MAX_INDEX`. When `++index` evaluates to `MAX_INDEX`, the **while** statement ends.

```
#define MAX_INDEX (sizeof(item) / sizeof(item[0]))
#include <stdio.h>
int main(void)
{
    static int item[ ] = { 12, 55, 62, 85, 102 };
    int index = 0;
    while (index < MAX_INDEX)
    {
        item[index] *= 3;
        printf("item[%d] = %d\n", index, item[index]);
        ++index;
    }
    return(0);
}
```

### Related Reading

“while”

---

## Statement Labels

A label is an identifier that allows your program to transfer control to other statements within the same function. It is the only type of identifier that has function scope.

Control is transferred to the statement following the label by means of the **goto** or **switch** statements. The **case** and **default** label names are reserved for use within the body of the **switch** statement.

▶ identifier : statement ◀

In the examples below, the label is both the identifier and the colon (:) character at the beginning of each line.

```
comment_complete : ;          /* null statement label */
test_for_null : if (NULL == pointer)
```

### Related Reading

“Scope of Identifier Visibility” on page 53

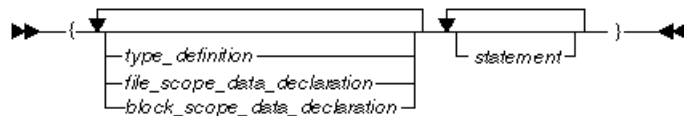
“goto” on page 151

“switch” on page 155

---

## Statement Blocks

A *block statement*, or *compound statement*, lets you group any number of data definitions, declarations, and statements into one statement. All definitions, declarations, and statements enclosed within a single set of braces are treated as a single statement. You can use a block wherever a single statement is allowed.



Definitions and declarations must come before the statements in a statement block.

Redefining a data object inside a nested block hides the outer object while the inner block runs. However, defining several variables that have the same identifier can make a program difficult to understand and maintain. You should avoid redefining identifiers within nested blocks. If a data object is usable within a block and its identifier is not redefined, all nested blocks can use that data object.

Initialization of an **auto** or **register** variable occurs each time the block is run from the beginning. If you transfer control from one block to the middle of another block, initializations are not always performed. You cannot initialize an **extern** variable within a block.

### Related Reading

“Block Scope Data Declarations” on page 62

“File Scope Data Declarations” on page 63

“Storage Class Specifiers” on page 65

“Type Specifiers” on page 66

“Example of Initialization within Statement Blocks”

## Example of Initialization within Statement Blocks

The following program shows how the values of data objects change in nested statement blocks:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int x = 1;          /* Initialize x to 1 */
```

```

6     int y = 3;
7
8     if (y > 0)
9     {
10        int x = 2;           /* Initialize x to 2 */
11        printf("second x = %4d\n", x);
12    }
13    printf("first  x = %4d\n", x);
14 }

```

The program produces the following output:

```

second x =    2
first  x =    1

```

Two variables named `x` are defined in `main`. The definition of `x` on line 5 retains storage while `main` is running. However, because the definition of `x` on line 10 occurs within a nested block, line 11 recognizes `x` as the variable defined on line 10. Because line 13 is not part of the nested block, `x` is recognized as the variable defined on line 5.

### Related Reading

“Initializers” on page 65

“Statement Blocks” on page 159

---

## C Programming Character Set

You can use any of the following characters from the ASCII character set to enter programming text into your source file.

- ```

a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
- ```

0 1 2 3 4 5 6 7 8 9

```
- ```

! " # $ % & ' ( ) * + , - . / :
; < = > ? [ \ ] ^ _ { | }

```

- The space character.
- The control characters representing horizontal tab, vertical tab, and form feed.

Some characters from the C character set are not available in all environments. You can enter these characters into a C source file using a sequences of two or three characters. A sequence of three characters called a *trigraph*. A sequence of two characters is called a *digraph*, but will be accepted by the compiler only if the **-qdigraph** compiler option is in effect. Digraph or trigraph character sequences appearing in character or string literals are not replaced during the preprocessor stage.

Digraph and trigraph sequences available to you are:

| C for AIX Digraph and Trigraph Sequences |           |                          |               |
|------------------------------------------|-----------|--------------------------|---------------|
| Digraphs                                 | Trigraphs | Character(s) Represented | Description   |
| %%                                       | ??=       | #                        | pound sign    |
| <:                                       | ??(       | [                        | left bracket  |
| :<                                       | ??)       | ]                        | right bracket |
| <%                                       | ??<       | {                        | left brace    |

|     |     |      |                                                                   |
|-----|-----|------|-------------------------------------------------------------------|
| %>  | ??> | }    | right brace                                                       |
|     | ??/ | \    | backslash                                                         |
|     | ??' | ^    | caret                                                             |
|     | ??! |      | pipe                                                              |
|     | ??- | ~    | tilde                                                             |
| %%: |     | /**/ | substitute for ##<br>preprocessor macro<br>concatenation operator |

#### Related Reading

“digraph” on page 252

---

## Escape Sequences for Non-Printable Characters

| Escape Sequence | Character Represented |
|-----------------|-----------------------|
| \a              | Alert (bell, alarm)   |
| \b              | Backspace             |
| \f              | Form feed (new page)  |
| \n              | New-line              |
| \r              | Carriage return       |
| \t              | Horizontal tab        |
| \v              | Vertical tab          |
| \'              | Single quotation mark |
| \"              | Double quotation mark |
| \?              | Question mark         |
| \\              | Backslash             |

#### Related Reading

“Escape Sequences” on page 52

---

## Reserved Keywords

Keywords are identifiers reserved by the C language for special use. You can use them for preprocessor macro names but that is, however, considered poor programming style.

Only the exact character case and spelling of keywords is reserved. For example, **auto** is reserved, but **AUTO** is not.

Keywords reserved by the C programming language are:

|        |          |          |        |
|--------|----------|----------|--------|
| auto   | break    | case     | char   |
| const  | continue | default  | do     |
| double | else     | enum     | extern |
| float  | for      | goto     | if     |
| int    | long     | register | return |
| short  | signed   | sizeof   | static |
| struct | switch   | typedef  | union  |

unsigned

void

volatile

while

Identifier names should not start with an underscore (`_`) followed by an uppercase letter, and should not contain two underscores (`__`) anywhere. The compiler and library use identifiers beginning with single and double underscores for their own purposes.

---

## Differences Between C Language Levels

### Conflicts Between extended C and Other Levels

**extended** level C, the default language level for the **cc** compiler invocation command, adheres to the ANSI/ISO C definition except where adherence conflicts with compatibility with the RT C implementation. In the case of certain obsolete RT C language definitions, adherence to the ANSI/ISO C standard takes precedence over compatibility with RT C.

This page lists the conflicts between:

- **extended** C and **ansi** C (page 162)
- **extended** C and RT C (page 163)

#### Conflicts Between extended C and ansi C

The following are areas where **extended** level C supports RT C constructs and conventions not supported by **ansi** level C:

- Macro expansion within quoted strings (either single or double quotation marks):  
Macro parameters found within quoted strings in replacement text are not replaced in **ansi** mode but are replaced in **extended** mode.
- Arithmetic conversions (for example, integral promotions):  
**extended** level C follows the rules outlined in “Arithmetic Conversions for extended Level C” on page 167, which differ from the **ansi** level C conversion rules defined in section 3.2.1.5 of the ANSI/ISO C Standard.
- Scope of external functions declared at block scope:  
In **ansi** mode, external functions have block scope. In **extended** mode, they have file scope.
- Implicit pointer conversions:  
**extended** mode allows assignment of a pointer to an object of a different type. In **ansi** mode, a cast operation is necessary if conversion is to be done.
- Pointers of different types can be assigned to each other:  
In **ansi** mode, an attempt to assign pointers of different types to each other produces an error message. **extended** mode accepts the assignment.
- **enum** declarations containing a trailing extra comma:  
Accepted in **extended** mode but not in **ansi** mode.
- No definition of static function:  
**ansi** mode requires a function declared **static** to be defined in the same file as the declaration. In **extended** mode, a function that is declared **static** but is not defined in the same file as the declaration is implicitly redeclared **extern**. An informational message is produced.
- **sizeof** operator on bit fields:  
Allowed by **extended** but not by **ansi**.
- String literals:

Both **ansi** and **extended** modes give you the choice of making string literals either modifiable or unmodifiable, but the defaults are different. The default is modifiable for **extended** and unmodifiable for **ansi**.

- Ref/def model:

The ANSI/ISO C standard defines a relaxed, strict, and initialization ref/def model for objects with external linkage. **extended** mode supports the relaxed model, whereas **ansi** mode supports a combination of the strict and initialization models.

- **unsigned char** and **unsigned short** bit fields:

Allowed in **extended** mode but not in **ansi** mode. In **extended** mode, an error message is issued, but compilation continues. Both **unsigned char** and **unsigned short** are changed to **unsigned int**. For bit fields, **unsigned char** and **unsigned short** are changed to **unsigned int**.

- **long long int** type:

Allowed in **extended** mode but not in **ansi** mode.

- Character data types: **char**, **unsigned char**, or **signed char**:

In **ansi** mode the C compiler distinguishes between the three character types; in **extended** mode, the C compiler does not distinguish between **char** and **unsigned char**.

- Macro redefinition:

**ansi** level C requires that a macro be undefined before it can be defined again in a **#define** directive. **extended** level C allows macros to be redefined without first being undefined. An informational message is issued that states that the second definition is used.

- \$ (dollar sign) character in identifiers:

In **extended** mode only, the compiler allows the \$ (dollar sign) character to be used in identifier names to facilitate calls between different languages. The \$ (dollar sign) is not a valid character for identifiers in **ansi** mode.

- Macro concatenation using **/\*\*/**:

Allowed by **extended** but not by **ansi**.

### Conflicts Between extended C and RT C

Ideally, **extended** mode should handle all RT C source code without conflict. Certain obsolete RT C definitions do conflict with ANSI/ISO C. For these, **extended** mode follows ANSI/ISO C and conflicts with RT C.

**extended** level C does not support the following RT C definitions:

- **asm** and **fortran** keywords.
- External data with file scope, even though declared or defined at block scope.
- Array initialization without braces.
- The last member of a structure definition not terminated by a semicolon.
- Type specifier not required when a name that was previously defined as a **typedef** is redefined.
- **+=** and **-=** operators.
- **#ifdef** using logical operators or period (.).
- Taking the address of a register variable.
- Function declarations at file scope without type specifiers.
- Variable declarations without type specifiers.
- Structure definition containing an empty structure declaration.
- Predefined macro names **unix**, and **AIX**.

### Related Reading

- “C Language Levels” on page 78
- “Extensions to RT C Provided by extended C”
- “Exceptions to ansi C Addressed by classic C”
- “saal2 C Deviations from SAA Level 2 C” on page 167
- “Arithmetic Conversions for extended Level C” on page 167
- “Summary of C Language Level Conflicts” on page 170

## Extensions to RT C Provided by extended C

The following are part of the ANSI/ISO C definition and are part of the **extended** language level. They cause no conflict with existing RT C source:

- **#pragma**, **#elif**, and **#error** preprocessor directives.
- Ability to form macro literals using **#**.
- Macro concatenation using **##**.
- Recursive macro definitions are only expanded once.
- White space or comment allowed before **#**.
- Trigraph sequences.
- Redefinition of **typedef** names (variable defined as a **typedef** can be redeclared as an identifier).
- **const** and **volatile** type qualifiers.
- Support of the **signed** keyword with **char**, **int**, **short**, and **long** data types.
- Suffixes **l** and **L** for type **long double** floating-point constants.
- Suffixes **u** and **U** for types **unsigned char** and **unsigned int**.
- Hexadecimal constants of the form `\0xdd`.
- Unary **+** operator.
- **enum** and **void** types.
- Function prototypes (including variable number of arguments specified by an ellipsis (...)).
- Initialization of auto aggregate variables.

### Related Reading

- “C Language Levels” on page 78
- “Conflicts Between extended C and Other Levels” on page 162
- “Exceptions to ansi C Addressed by classic C”
- “saal2 C Deviations from SAA Level 2 C” on page 167
- “Arithmetic Conversions for extended Level C” on page 167
- “Summary of C Language Level Conflicts” on page 170

## Exceptions to ansi C Addressed by classic C

Exceptions to the **ansi** mode addressed by **classic** are as follows:

- Tokenization (page 164)
- Preprocessing Directives (page 165)
- Macro Expansion (page 166)
- Text Output

### Tokenization

Tokens introduced by macro expansion may be combined with adjacent tokens in some cases. Historically,



this effect was caused by 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:

- At a given point in the source file, the next token is the longest sequence of characters which could possibly form a token. For example, `i+++++j` is tokenized as `i ++ ++ + j` even though `i ++ + ++ j` is the intended tokenization.
- If the token formed is an identifier and it is 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.
- 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.
- 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 which 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 literals and character constants 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 `0`.

### 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 */
#define\
M 1         /* equivalent to #dfine 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):

|                         |                                                                                                                                                                                                                                                                                                          |
|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#ifndef/#ifnndef</b> | When the first token is not an identifier, no diagnostic message is generated, and the condition is FALSE.                                                                                                                                                                                               |
| <b>#else</b>            | When there are extra tokens, no diagnostic message is generated.                                                                                                                                                                                                                                         |
| <b>#endif</b>           | When there are extra tokens, no diagnostic message is generated.                                                                                                                                                                                                                                         |
| <b>#include</b>         | 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 that " and ' do begin literals within the < and >.                                             |
|                         | <b>Note:</b> In C programs, //-style comments are only valid when <b>cpluscmt</b> is specified.)                                                                                                                                                                                                         |
| <b>#line</b>            | 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.                                                                                                                                                                       |
| <b>#error</b>           | Not recognized in <b>classic</b> mode.                                                                                                                                                                                                                                                                   |
| <b>#define</b>          | A valid macro parameter list consists of zero or more identifiers each separated by zero or more 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 honored. |
|                         | 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.                                                                                                                                            |
| <b>#undef</b>           | 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 */

```

No text is generated to replace comments.

### Related Reading

- “C Language Levels” on page 78
- “Conflicts Between extended C and Other Levels” on page 162
- “Extensions to RT C Provided by extended C” on page 164
- “saal2 C Deviations from SAA Level 2 C”
- “Arithmetic Conversions for extended Level C”
- “Summary of C Language Level Conflicts” on page 170
- “cplusplus” on page 247

## saal2 C Deviations from SAA Level 2 C

**saal2** level C deviates from the SAA C definition as follows:

- The **\_Packed** attribute for structures and unions is not supported by the C compiler component of C for AIX. The **-qalign=packed** compiler option provides some of the function of the attribute.
- Record input/output is not supported by the AIX Version 3.2 operating system, and is not available on the C compiler. SAA Level 2 defines record input/output.
- AIX Version 3.2 operating system error conditions for the following differ from those of SAA Level 2:

|      |       |       |                                   |
|------|-------|-------|-----------------------------------|
| acos | asin  | atan2 | the bessel functions (y0, y1, yn) |
| fmod | gamma | log   | log10                             |

pow sq

### Related Reading

- “C Language Levels” on page 78
- “Conflicts Between extended C and Other Levels” on page 162
- “Extensions to RT C Provided by extended C” on page 164
- “Exceptions to ansi C Addressed by classic C” on page 164
- “Arithmetic Conversions for extended Level C”
- “Summary of C Language Level Conflicts” on page 170
- “align” on page 234

## Arithmetic Conversions for extended Level C

This page describes the rules for arithmetic conversions that the compiler adheres to when the language specified is **extended**. Described are:

- Usual Unary Conversions (page 167)
- Usual Arithmetic Conversions (page 168)
  - Widening (page 168)
  - Type Balancing (page 168)
  - Sign Balancing (page 169)
- Assignment Conversions Table (page 169)
- Explicit Conversions (page 169)
  - Reduction Conversions (page 169)
  - Expansion Conversions (page 170)
  - Pointer Conversions (page 170)
  - void Conversions (page 170)
  - volatile Conversions (page 170)

**Usual Unary Conversions**The usual unary conversions reduce the types of values that the compiler must handle. The compiler uses the usual unary conversions on:

- The operands of the unary operators: `!`, `-`, `~`, and `*`

- The operands of the binary operators < and >
- The arguments in a function call (before the function is called and if a function prototype is not available).

The following table lists the types of values that the usual unary conversions affect:

| Type of Value...  |                  |
|-------------------|------------------|
| before Conversion | after Conversion |
| char              | int              |
| unsigned char     | unsigned int     |
| short             | int              |
| unsigned short    | unsigned int     |
| float             | double           |
| array of type     | pointer to type  |

**Note:** The compiler performs the usual unary conversion of **float** to **double** on arguments in function calls only. When a **float** object appears as an operand of **!**, **-**, **~**, **\***, **<**, or **>**, the compiler does not perform a usual unary conversion.

Explicit Conversions (page 169) describes how the compiler performs conversions.

**Usual Arithmetic Conversions** The usual arithmetic conversions reduce the types of objects that the compiler handles when performing arithmetic operations. Many compilers perform arithmetic operations only on objects having one of several data types. These types are: **int**, **unsigned int**, **long**, **unsigned long**, **float**, **double**, and **long double**. If all operands do not have one of these types, the system converts the values of the operands according to the following procedures:

1. *Widening* values that do not have data types appropriate for arithmetic operations.
2. *Type balancing* values in operations that have more than one operand.
3. *Sign balancing* values in operations that have more than one operand.

The following sections describe the usual arithmetic conversion procedures.

**Widening** *Widening* expands the size of a value (for example, **short** to **int** by padding bits located to the left of the value with a copy of the sign bit). Widening does not affect the sign of the value.

The following table shows the types of values that the compiler widens:

| Type of Value... |                     |
|------------------|---------------------|
| before Widening  | after Widening      |
| char             | int OR unsigned int |
| unsigned char    | unsigned int        |
| short            | int                 |
| unsigned short   | unsigned int        |
| float            | double              |

The compiler treats **char** objects as unsigned values. Widening of a **char** yields an **int** that has a positive value.

Many compilers widen **float** values to **double** values before performing arithmetic operations. Where possible, C for AIX performs double-precision arithmetic on **float** values.

## Type Balancing

*Type balancing* makes all operands have the same data type. If both of the operands do not have the same size data type, the compiler converts the value of the operand having the smaller type to a value having the larger type. For example, if the operand `count` has type `int` and the operand `maximum` has type `long`, the compiler converts the value of `count` to type `long`.

Type balancing does not affect the sign of the value.

**Sign Balancing** *Sign balancing* makes both operands have the same data type (**signed** or **unsigned**). If one operand has an **unsigned** type, the compiler converts the other operand to that **unsigned** type. Otherwise, both operands remain signed.

**Assignment Conversion** An assignment conversion makes the value of the right operand have the same data type as the left operand. Only the following assignment type combinations are supported by the language:

| Type of...               |                                               |
|--------------------------|-----------------------------------------------|
| Left Operand             | Right Operand                                 |
| Any arithmetic type      | Any arithmetic type                           |
| Pointer to <i>type</i>   | Pointer to <i>type</i> , or, the NULL pointer |
| Structure of <i>type</i> | Structure of <i>type</i>                      |
| Union of <i>type</i>     | Union of <i>type</i>                          |

Explicit Conversions (page 169) describes how the compiler performs conversions from one arithmetic type to another arithmetic type.

**Explicit Conversions** When the compiler converts the values of one data type to the value of another data type, the compiler usually performs one of the following conversions:

|                              |                                                                                                                                                    |
|------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Reduction conversions</b> | Change the data type of a value to a smaller size data type (for example, a value having type <b>double</b> to a value having type <b>float</b> ). |
| <b>Expansion conversions</b> | Change the data type of a value to a larger size data type (for example, a value having type <b>float</b> to a value having type <b>double</b> ).  |
| <b>Pointer conversions</b>   | Change the data type to which a pointer refers or change an integral type to a pointer.                                                            |
| <b>void conversions</b>      | Discard the value of a function call.                                                                                                              |
| <b>volatile conversions</b>  | Give a nonvolatile data object the <b>volatile</b> attribute.                                                                                      |

The following sections describe these conversions.

## Reduction Conversions

|                                       |                                                                                                                                                                                                                                               |
|---------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Integral Reduction</b>             | The compiler converts an integral value to a narrower type (for example, a <b>long</b> to a <b>short</b> ); the compiler truncates the value by discarding the most significant bits.                                                         |
| <b>double or long double to float</b> | The compiler converts a double-precision floating-point value ( <b>long double</b> or <b>double</b> ) to a single-precision floating-point value ( <b>float</b> ); the compiler rounds off the double-precision value.                        |
| <b>Floating-Point to Integral</b>     | The language does not define the method of converting floating-point values to integral values. The compiler drops the fraction part of the floating-point value.                                                                             |
| <b>Integral to Floating-Point</b>     | The C language does not prohibit integral sizes from having a higher precision than the floating-point sizes. If a higher precision integer is converted to a <b>float</b> , the resulting <b>float</b> might experience a loss of precision. |

## Expansion Conversions

### Floating-Point Expansion

Although many compilers perform all floating-point arithmetic in double precision only, the C compiler **extended** language level performs double-precision arithmetic when all operands have type **float**. When one operand has type **float** and another operand has type **double** or **long double**, the compiler converts the **float** to the equivalent **double** or **long double** value.

### Integral to Floating-Point

The compiler converts narrower integral values to equivalent floating-point values.

### Unsigned Arithmetic Expansion

The compiler converts narrower **unsigned** arithmetic values to wider **unsigned** arithmetic values by padding the values with zeros.

### Signed Arithmetic Expansion

The language does not define how narrower signed arithmetic values are converted to wider signed arithmetic values. When a narrower arithmetic value is converted to a wider signed arithmetic value, the compiler pads bits located to the left of the value with a copy of the sign bit.

## Pointer Conversions

### Pointer to Pointer

When two pointers to objects of the same type are subtracted, the compiler performs the operation on the values of the pointers and divides the result by the length of the objects to which the pointers refer. The result is an integer that indicates the distance between the specified objects in the array. For example, if **p** points to the second element in an array and **q** points to the fifth element in the array, the expression **p - q** yields -3.

### Integral to Pointer

When an integral value is subtracted from a pointer, the compiler multiplies the integral value by the length of the object to which the pointer refers to produce an address offset, which can be added or subtracted from the pointer value. The result is a pointer (having the same type as the original pointer) that refers to an object assumed to be in the same array.

**void Conversions** A program cannot use or apply conversions to the (nonexistent) value of a **void** object. To convert the result of a function call to type **void**, use the cast operator. Such a conversion discards the value of a function call used in an expression statement. For example, the following statement discards the result of the function call `add()`:

```
(void)add();
```

## volatile Conversions

### volatile to Nonvolatile

Through an explicit cast, you can assign the address of a **volatile** data object to a pointer that is defined as pointing to a nonvolatile data object. If the **volatile** object is referenced through such a pointer, the result is undefined.

### Nonvolatile to volatile

You can assign the address of a nonvolatile data object that is defined as pointing to a **volatile** data object. If the nonvolatile object is referenced through such a pointer, the compiler treats the nonvolatile object as a **volatile** object.

## Related Reading

“C Language Levels” on page 78

“Arithmetic Conversions Table” on page 137

“Conflicts Between extended C and Other Levels” on page 162

“Extensions to RT C Provided by extended C” on page 164

“Exceptions to ansi C Addressed by classic C” on page 164

“saal2 C Deviations from SAA Level 2 C” on page 167

“Summary of C Language Level Conflicts”

## Summary of C Language Level Conflicts

This section summarizes for quick reference the conflicts listed in the related pages.

**Note:** The following table shows only those features of **extended** level C that conflict with either RT C or ANSI/ISO C. Features that are part of **extended** C but not part of RT C are not listed in the table unless they present a conflict.

| Area of Conflict                                                                              | RT C          | ansi C        | extended C                     |
|-----------------------------------------------------------------------------------------------|---------------|---------------|--------------------------------|
| <b>asm</b> and <b>fortran</b> keywords                                                        | Supported     | Not Supported | Not Supported                  |
| <b>=+</b> and <b>=-</b> operator                                                              | Supported     | Not Supported | Not Supported                  |
| Type specifier not required when a name that was previously defined as a typedef is redefined | Supported     | Not Supported | Not Supported                  |
| Scope of external data declared or defined at block scope                                     | File          | Block         | Block                          |
| <b>#ifdef</b> using logical operators or period (.)                                           | Supported     | Not Supported | Not Supported                  |
| Taking the address of a register variable                                                     | Supported     | Not Supported | Not Supported                  |
| Modifiable or unmodifiable string literals                                                    | Modifiable    | Unmodifiable  | Modifiable                     |
| Relaxed ref/def model                                                                         | Supported     | Not Supported | Supported                      |
| <b>unsigned char</b> and <b>unsigned short</b> bit fields                                     | Supported     | Not Supported | Changed to <b>unsigned int</b> |
| <b>unsigned char</b> and <b>char</b> recognized as incompatible types                         | Not Supported | Supported     | Not Supported                  |
| <b>long long int</b> type                                                                     | Not Supported | Not Supported | Supported                      |
| Scope of external functions declared at block scope                                           | File          | Block         | File                           |
| Handling of macro parameters within string literals                                           | Expanded      | Not Expanded  | Expanded                       |
| Preprocessor macro can be redefined without first being undefined                             | Supported     | Not Supported | Supported                      |
| Rules followed when performing arithmetic conversions                                         | RT C          | ANSI/ISO      | RT C                           |
| All valid pointer conversions without an explicit cast                                        | Supported     | Not Supported | Supported                      |
| Assignment of pointers to different types                                                     | Supported     | Not Supported | Supported                      |
| Enumeration declarations with trailing extra comma                                            | Supported     | Not Supported | Supported                      |
| Functions without definition accepted and defined extern                                      | Supported     | Not Supported | Supported                      |
| Definition of static function                                                                 | Not Required  | Required      | Not Required                   |
| <b>sizeof</b> operator on bit fields                                                          | Supported     | Not Supported | Supported                      |

|                                                                     |                     |               |               |
|---------------------------------------------------------------------|---------------------|---------------|---------------|
| Array initialization without braces                                 | Supported           | Not Supported | Not Supported |
| Last member of a structure definition not terminated by a semicolon | Supported           | Not Supported | Not Supported |
| Predefined macro name                                               | <b>unix and AIX</b> | <b>_AIX</b>   | <b>_AIX</b>   |
| \$ character in identifiers                                         | Permitted           | Prohibited    | Permitted     |
| Macro concatenation using <b>/**/</b>                               | Supported           | Not Supported | Supported     |

### Related Reading

“C Language Levels” on page 78

“Conflicts Between extended C and Other Levels” on page 162

“Extensions to RT C Provided by extended C” on page 164

“Exceptions to ansi C Addressed by classic C” on page 164

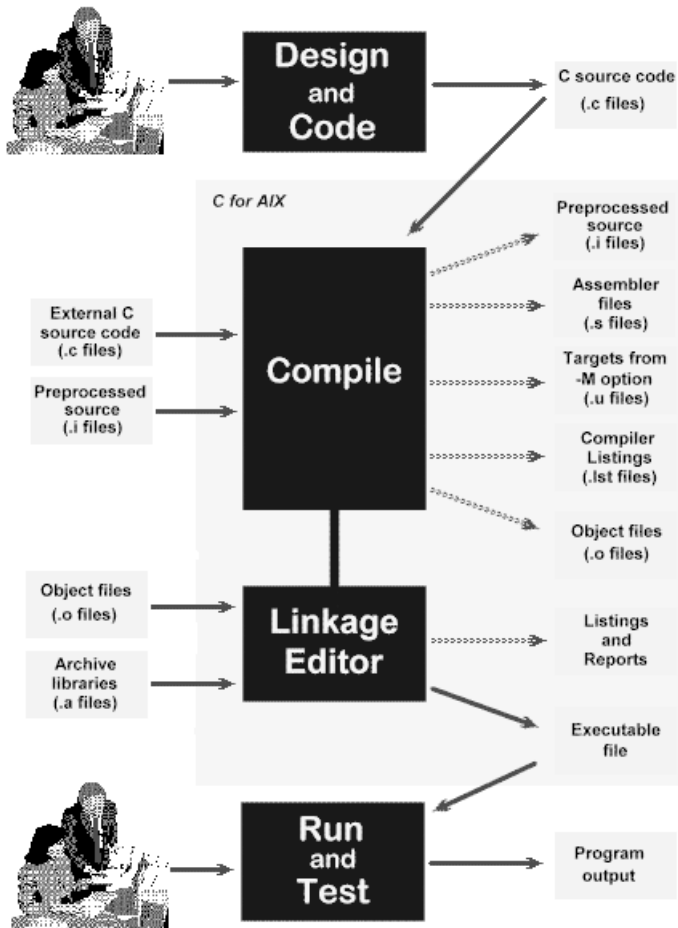
“saal2 C Deviations from SAA Level 2 C” on page 167

“Arithmetic Conversions for extended Level C” on page 167



# Chapter 7. Writing C Programs

A C program typically passes through four steps of development.



The solid lines show inputs into each step of the development cycle. Compile and Linkage Editor operations are performed by the C for AIX product, which also lets you specify what optional outputs are produced. Optional outputs are shown in the diagram by the broken lines. Descriptions of the steps follow below:

- Design and Code** Involves designing a program to meet a specified requirement, and creating the programming language text files that will comprise the program source.
- Compile** After checking for syntactical correctness, converts the programming language source files into machine readable instructions, where C variables are associated with memory addresses, and C statements are turned into a series of machine language instructions. The compiler can produce various forms of output, depending on the compiler options selected.
- Linkage Editor** Links compiler output with external modules requested by the compiled program. C programs can use routines from C libraries or any object or archive file from the IBM XL family of languages. C programs can also use modules produced by the current or previous compilations. As well as linking the external modules, the linkage editor resolves addresses within the object module.
- Run and Test** This stage can be both the final step in program development, or it can be an intermediate point in the program design and implementation process. A program's design commonly is further refined as a result of information gathered during testing.

## Related Reading

- “Creating and Naming a C Source File”
- “Internal Structure of a C Program”
- “External Structure of a C Program” on page 177
- “Writing Optimized Program Source Code” on page 197
- “Compiler Options and Their Defaults” on page 218
- “Message Severity Levels and Compiler Response” on page 20
- “Compiler Return Codes” on page 20
- “Compiler Message Format” on page 21

---

## Creating and Naming a C Source File

A C program source is a collection of one or more text source files written in the C programming language, each of which can contain all or part of the functions that make up a C program. The individual source files are compiled into object modules which can then be linked together to create an executable program. You can use any text editor to create and edit a source file.

## File-Naming Conventions

A file name can be up to 256 characters. (Longer names are truncated on the right.) The file name can contain lowercase and uppercase letters, numbers, underscores, periods, and other characters. The AIX Version 4 Operating System distinguishes between uppercase and lowercase letters.

By convention, C source files end with a `.c` filename extension, for example,

```
myprogram.c
```

The characters `& | ; ( ) < > ? / * ' `` have special meaning in the AIX system. To use them in a file name, you must place them inside quotation marks so that the shell does not interpret them. For example:

```
my"<"new">"program.c
```

## Related Reading

- “Chapter 7. Writing C Programs” on page 173
- “Internal Structure of a C Program”
- “External Structure of a C Program” on page 177
- “C Language Levels” on page 78
- “Writing Optimized Program Source Code” on page 197
- “C Programming Character Set” on page 160
- “Reserved Keywords” on page 161
- “Appendix M. ASCII Character Set” on page 483

## Internal Structure of a C Program

A C source program is a collection of one or more directives, declarations, and statements contained in one or more source files.

|                     |                                                                                                                                                                          |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>statements</b>   | Specify the action to be performed.                                                                                                                                      |
| <b>directives</b>   | Instruct the preprocessor to act on the text of the program. Pragma directives affect compiler behavior.                                                                 |
| <b>declarations</b> | Establish names and define linkage characteristics such as scope, data type, and linkage.                                                                                |
| <b>definitions</b>  | Are declarations that allocate storage for data objects or define a body for functions. An object definition allocates storage and may optionally initialize the object. |

A function declaration precedes the function body. The function body is a compound statement that can contain declarations and statements that define what the function does. The function declaration declares the name, its parameters, and the data type of the value it returns.

A program must contain at least one function declaration. If the program contains only one function declaration, the function must be called **main**. If the program contains more than one function declaration, only one of the functions can be called **main**. Any additional functions called **main** are ignored.

By convention, **main** is the starting point for running a program. The main function can in turn call other functions. A program usually stops running at the first encounter of any of the following:

- The end of the **main** function
- A **return** statement in the **main** function
- An **exit** function call

A C program can contain any number of directives, declarations, and definitions. Before the C program is compiled, the preprocessor filters out preprocessor directives that may change the files. Preprocessor directives are completed, macros are expanded, and a temporary source file is created containing C statements, completed directives, declarations, and definitions.

It is sometimes useful to gather variable definitions into one source file and declare references to those variables in any source files that use them. This procedure makes definitions easy to find and change. You can also organize constants and macros into separate files, and include them into source files as needed. You can use the **#include** directive to imbed such source files into other source files.

Directives in a source file apply to that source file and its included files only. Each directive applies only to the part of the file (and included files) following the directive.

The C for AIX compiler looks for a function called **main** in the source code and uses it as the entry point name. If a program contains more than one function definition, only one of these functions can be named **main**. If the program contains only one function definition, that function must be called **main**.

#### **Related Reading**

- “External Structure of a C Program” on page 177
- “Scope of Identifier Visibility” on page 53
- “Statement Blocks” on page 159
- “Example of a Simple C Program”
- “Example of a C Program Comprised of Two Source Files” on page 176
- “Specifying Path Names for Include Files” on page 178
- “C Programming Character Set” on page 160
- “Reserved Keywords” on page 161
- “Appendix M. ASCII Character Set” on page 483

## **Example of a Simple C Program**

The source for a simple C program is shown below:

### **A Simple C Program**

```

/**
 ** This is an example of a simple C program
 **/
#include <stdio.h>    /* standard I/O library header that
                    contains macros and function
                    declarations, ie printf used below */
#include <math.h>    /* standard math library header that
                    contains macros and function
                    declarations, ie cos used below */
#define NUM 46.0    /* Preprocessor directive */
double x = 45.0;    /* External variable definitions */
double y = NUM;
int main(void)      /* Function definition
                    for main function */
{
    double z;      /* Local variable definitions */
    double w;
    z = cos(x);    /* cos is declared in math.h as
                    double cos(double arg) */

    w = cos(y);
    printf ("cosine of x is %f\n", z); /* Print cosine of x */
    printf ("cosine of y is %f\n", w); /* Print cosine of y */
    return 0;
}

```

The program above defines **main** and declares a reference to the function **cos**. The program defines the global variables **x** and **y**, initializes them, and declares two local variables **z** and **w**.

### Related Reading

- “Internal Structure of a C Program” on page 174
- “Scope of Identifier Visibility” on page 53
- “Statement Blocks” on page 159
- “Example of a C Program Comprised of Two Source Files”

## Example of a C Program Comprised of Two Source Files

The following example shows a C program source comprised of two source files. The **main** and **max** functions are in separate files. The program logic starts with the **main** function.

### Example Program with Two Source Files

```

/*****
 * Source file 1 - main function
 *****/
#define ONE 1
#define TWO 2
#define THREE 3
extern int max(int, int); /* Function declaration */
int main(int argc, char * argv[]) /* Function definition */
{
    int u, w, x, y, z;
    u = 5;
    z = 2;
    w = max(u, ONE);
    x = max(w, TWO);
    y = max(x, THREE);
    z = max(y, z);
    return z;
}

```

```

/*****
* Source file 2 - max function
*****/
int max (int a,int b)      /* Function definition */
{
    if ( a > b )
        return (a);
    else
        return (b);
}

```

The first source file declares the function **max**, but does not define it. This is an *external declaration*, a declaration of a function defined in source file 2. Four statements in **main** are *function calls* of **max**.

The lines beginning with a number sign (#) are preprocessor directives that direct the preprocessor to replace the identifiers ONE, TWO, and THREE with the digits 1, 2, and 3. The directives in the first source file do not apply to the second source file.

The second source file contains the function definition for **max**, which is called four times in **main**. After you compile the source files, you can link and run them as a single program.

### Related Reading

“Internal Structure of a C Program” on page 174

“Scope of Identifier Visibility” on page 53

“Statement Blocks” on page 159

“Example of a Simple C Program” on page 175

## External Structure of a C Program

A source program consists of at least one source file. You can compile a source program that consists of several source files by specifying all of the source files as input to the compiler invocation command. Typically, compiler invocation produces calls to both the compiler and the linkage editor, and creates a single executable file as output. For example, to produce an executable file named **testprog** from three files, **testdata.c**, **testres.c**, and **testparm.c**, you would enter:

```
xlc testdata.c testres.c testparm.c -o testprog
```

You can also compile each source file separately by specifying the “**c**” on **page 242** compiler option to invoke only the compiler to produce object files (**.o** files). You can then link-edit the resulting object files to create an executable file by invoking the compiler on these **.o** files without using the “**c**” on **page 242** option.

For example, to produce object files for each of three programs, **testdata.c**, **testres.c**, and **testparm.c**, you would enter:

```
xlc testdata.c testres.c testparm.c -c
```

Then, to produce an executable file named **testprog** from these three object files, **testdata.o**, **testres.o**, and **testparm.o**, enter:

```
xlc testdata.o testres.o testparm.o -o testprog
```

To combine several source files at compilation, you can list the files on the command line when you use an invocation command to produce a compiled file for each file you specify. Or you can use the “**#include Preprocessor Directive**” on **page 356** preprocessor directive to include the files in the primary source file so that one compiled file is produced. This directive causes the text of a named secondary source file to be imbedded at the point where the **#include** is encountered in the primary file.

## Related Reading

“Internal Structure of a C Program” on page 174

“Specifying Path Names for Include Files”

“Compiler Options and Their Defaults” on page 218

---

## Specifying 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.

### Using 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
```

### Using 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. See *AIX Version 4 System User's Guide: Operating System and Devices* for a complete explanation of the AIX file system.

## Directory Search Sequence for Include Files Using Relative Path Names

The C language defines two versions of the **#include** preprocessor directive. The C for AIX compiler 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                                                                                                                                                                                                                                                                                                                                                               |
|-----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#include</b> <file_name> | <ol style="list-style-type: none"><li>1. If you specify the <b>-Idirectory</b> 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.</li><li>2. Searches the directory <b>/usr/include</b>.</li></ol> |

|                                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|--------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>#include</b> “ <i>file_name</i> ” | <ol style="list-style-type: none"> <li>1. Searches the directory where your current source file resides. The current source file is the one that contains the directive <b>#include</b> “<i>file_name</i>”.</li> <li>2. If you specify the option <b>-Idirectory</b>, 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.</li> <li>3. Searches the directory <b>/usr/include</b>.</li> </ol> |
|--------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**Notes:**

1. *file\_name* is the path name of the file to be included. When 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. With a relative path name, the directory search sequence is determined by whether you use the < > or the “ ” characters.
2. The only difference between the two versions of the **#include** directive is that the “ ” (user include) version first searches in 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. The **/usr/include** directory is 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 Reading**

- “External Structure of a C Program” on page 177
- “I” on page 272
- “idirfirst” on page 273
- “stdinc” on page 325
- “I” on page 272

## Using Memory Heaps in a Program

### Memory Management Functions

The memory management functions defined by ANSI are `calloc`, `malloc`, `realloc`, and `free`. These regular functions allocate and free memory from the default runtime heap. C for AIX includes another function, `_heapmin`, to return unused memory to the system. C for AIX also provides enhanced versions of memory management functions that can help you improve program performance (link to the **libhm.a** library), work with user heaps, or debug your programs.

All the versions actually work the same way. They differ only in what heap they allocate from, and in whether they save information to help you debug memory problems. The memory allocated by all of these functions is suitably aligned for storing any type of object.

The table below summarizes the different versions of memory management functions, using `malloc` as an example of how the names of the functions change for each version.

|                   | Regular Version       | Debug Version               |
|-------------------|-----------------------|-----------------------------|
| Default Heap      | <code>malloc</code>   | <code>_debug_malloc</code>  |
| User-Created Heap | <code>_umalloc</code> | <code>_debug_umalloc</code> |

## Heap-Specific Functions

Use heap-specific versions of memory allocation functions to allocate and free memory from user-created heaps that you specify. If you want, you can also explicitly specify the runtime heap. The names of user-created heaps are prefixed by `_u` (for “user heaps”), for example, `_umalloc`, and they are defined in `<umalloc.h>`.

When working with user-created heaps, you need to link to the **libhu.a** library. Heap-specific functions provided in this library are:

- `_ucalloc`
- `_umalloc`
- `_uheapmin`

There are no heap-specific versions of `realloc` or `free`. These standard functions always determine which heap memory is allocated from, and can be used with both user-created and runtime memory heaps.

## Debug Functions

Use these functions to allocate and free memory from the default runtime heap, just as you would use the regular versions. They also provide information that you can use to debug memory problems.

Use the **-qheapdebug** compiler option to automatically map all calls to the regular memory management functions to their debug versions. You can also call the debug versions explicitly.

**Note:** If you parenthesize the calls to the regular memory management functions, they are **not** mapped to their debug versions.

You should place a **#pragma strings(readonly)** directive at the top of each source file that will call debug functions, or in a common header file that each includes. This directive is not essential, but it ensures that the file name passed to the debug functions can't be overwritten, and that only one copy of the file name string is included in the object module.

The names of the debug versions are prefixed by `_debug_`, for example, `_debug_malloc`, and they are defined in `<malloc.h>` and `<stdlib.h>`.

The functions provided are:

- `_debug_calloc`
- `_debug_free`
- `_debug_heapmin`
- `_debug_malloc`
- `_debug_realloc`

The **debug\_malloc**, **debug\_realloc**, and **debug\_free** functions set the memory areas they affect to a specific, repeating fill pattern. See “Debugging Memory Heaps” on page 183 for more information.

In addition to their usual behavior, these functions also store information (file name and line number) about each call made to them. Each call also automatically checks the heap by calling `_heap_check` (described below).

Three additional debug memory management functions do not have regular counterparts:



- `_dump_allocated`  
Prints information to **stderr** about each memory block currently allocated by the debug functions.
- `_dump_allocated_delta`  
Prints information to file handle 2 about each memory block allocated by the debug functions since the last call to `_dump_allocated` or `_dump_allocated_delta`.
- `_heap_check`  
Checks all memory blocks allocated or freed by the debug functions to make sure that no overwriting has occurred outside the bounds of allocated blocks or in a free memory block.

The debug functions call `_heap_check` automatically; and you can also call this function explicitly. The `_dump_allocated` and `_dump_allocated_delta` functions must be explicitly called.

## Heap-Specific Debug Functions

The heap-specific functions also have debug versions that work just like the regular debug versions. Use these functions to allocate and free memory from the user-created heap you specify, and also provide information that you can use to debug memory problems in your own heaps.

Use the **-qheapdebug** compiler option to automatically map all calls to the regular memory management functions to their debug versions. You can also call the debug versions explicitly.

**Note:** If you parenthesize the calls to the regular memory management functions, they are **not** mapped to their debug versions.

The names of the heap-specific debug versions are prefixed by `_debug_u`, for example, `_debug_umalloc`, and they are defined in `<umalloc.h>`.

The functions provided are:

- `_debug_ucalloc`
- `_debug_uheapmin`
- `_debug_umalloc`
- `_udump_allocated`
- `_udump_allocated_delta`
- `_uheap_check`

The **debug\_umalloc** function sets the memory areas they affect to a specific, repeating fill pattern. See “Debugging Memory Heaps” on page 183 for more information.

There are no heap-specific debug versions of `_debug_realloc` or `_debug_free`. These functions always determine which heap memory is allocated from, and can be used with both user-created and runtime memory heaps.

## Related Reading

“Managing Memory with Multiple Heaps”

“Types of Memory” on page 183

“Debugging Memory Heaps” on page 183

“Creating and Using a Fixed Size Heap” on page 186

“Creating and Using an Expandable Heap” on page 188

“Debugging Programs with Heap Memory” on page 195

“Changing the Default Heap Used in a Program” on page 185

“Example of Creating and Using a User Heap” on page 190

“Example of Creating and Using a Shared-Memory User Heap” on page 191

“#pragma strings Preprocessor Directive” on page 376

“heapdebug” on page 270

## Managing Memory with Multiple Heaps

C for AIX lets you create and use your own pools of memory, called *heaps*. You can use your own heaps in place of or in addition to the default C for AIX runtime heap to improve the performance of your program.

**Note:** Many readers will not be interested in creating their own heaps. Using your own heaps is entirely optional, and your applications will work perfectly well using the default memory management provided (and used by) the C for AIX runtime library. If you want to improve the performance and memory management of your program, multiple heaps can help you. Otherwise, you can ignore this section and any heap-specific library functions.

## Why Use Multiple Heaps?

Using a single runtime heap is fine for most programs. However, using multiple heaps can be more efficient and can help you improve your program’s performance and reduce wasted memory for a number of reasons:

- When you allocate from a single heap, you may end up with memory blocks on different pages of memory. For example, you might have a linked list that allocates memory each time you add a node to the list. If you allocate memory for other data in between adding nodes, the memory blocks for the nodes could end up on many different pages. To access the data in the list, the system may have to swap many pages, which can significantly slow your program.

With multiple heaps, you can specify which heap you allocate from. For example, you might create a heap specifically for the linked list. The list’s memory blocks and the data they contain would remain close together on fewer pages, reducing the amount of swapping required.

- In multithread applications, only one thread can access the heap at a time to ensure memory is safely allocated and freed. For example, say thread 1 is allocating memory, and thread 2 has a call to free. Thread 2 must wait until thread 1 has finished its allocation before it can access the heap. Again, this can slow down performance, especially if your program does a lot of memory operations.

If you create a separate heap for each thread, you can allocate from them concurrently, eliminating both the waiting period and the overhead required to serialize access to the heap.

- With a single heap, you must explicitly free each block that you allocate. If you have a linked list that allocates memory for each node, you have to traverse the entire list and free each block individually, which can take some time.

If you create a separate heap for that linked list, you can destroy it with a single call and free all the memory at once.

- When you have only one heap, all components share it (including the C for AIX runtime library, vendor libraries, and your own code). If one component corrupts the heap, another component might fail. You may have trouble discovering the cause of the problem and where the heap was damaged.

With multiple heaps, you can create a separate heap for each component, so if one damages the heap (for example, by using a freed pointer), the others can continue unaffected. You also know where to look to correct the problem.

You can create heaps of regular memory or shared memory, and you can have any number of heaps of any type. See “About this Information” on page xvii for more information about the different types of memory for heaps. The only limit is the space available on your operating system (your machine’s memory and swapper size, minus the memory required by other running applications).

C for AIX provides heap-specific versions of the memory management functions, for example, `umalloc` and so on. Debug versions of all memory management functions are provided, including the heap-specific ones. C for AIX also provides additional functions that you can use to create and manage your own heaps of memory, such as **udefault**.

#### **Related Reading**

“Memory Management Functions” on page 179

“Types of Memory”

“Debugging Memory Heaps”

“Creating and Using a Fixed Size Heap” on page 186

“Creating and Using an Expandable Heap” on page 188

“Debugging Programs with Heap Memory” on page 195

“Changing the Default Heap Used in a Program” on page 185

“Example of Creating and Using a User Heap” on page 190

“Example of Creating and Using a Shared-Memory User Heap” on page 191

## **Types of Memory**

There are two types of memory:

1. Regular memory

Most programs use regular memory. This is the type provided by the default runtime heap.

2. Shared memory

Heaps of shared memory can be shared between processes or applications. If you want other processes to use the heaps you have created, you must pass them the heap handle and give them access to the heap. Use `_ucreate` to create the heap.

#### **Related Reading**

“Memory Management Functions” on page 179

“Managing Memory with Multiple Heaps” on page 182

“Debugging Memory Heaps”

“Example of Creating and Using a User Heap” on page 190

“Example of Creating and Using a Shared-Memory User Heap” on page 191

## **Debugging Memory Heaps**

C for AIX provides two sets of functions for debugging your memory problems:

1. Debug versions of all memory management functions

2. Heap-checking functions similar to those provided by other compilers.

## **Debug Memory Management Functions**

Debug versions of the heap-specific memory management functions are provided, just as they are for the regular versions. Each debug version performs the same function as its non-debug counterpart. In

addition, the debug version calls `_uheap_check` to check the heap used in the call, and records the file and line number where the memory was allocated or freed. You can then use `_dump_allocated` or `_dump_allocated_delta` to display information about currently allocated memory blocks. Information is printed to **stderr**.

You can use debug memory management functions for any type of heap, including shared memory. To use the debug versions, specify the **-qheapdebug** compiler option. The C for AIX compiler then maps all calls to memory management functions (regular or heap-specific) to the corresponding debug versions.

**Note:** If you parenthesize the name of a memory management function, the function is **not** mapped to the debug version.

## Heap-Checking Functions

C for AIX also provides some new functions for validating user heaps: `_uheapchk`, `_uheapset`, and `_uheap_walk`. Each of these functions also has a non-heap-specific version that validates the default heap.

Both `_uheapchk` and `_uheapset` check the specified heap for minimal consistency; `_uheapchk` checks the entire heap, while `_uheapset` checks only the free memory. `_uheapset` also sets the free memory in the heap to a value you specify. `_uheap_walk` traverses the heap and provides information about each allocated or freed object to a callback function that you provide. You can then use the information however you like.

These heap-checking functions are defined in **<umalloc.h>** (the regular versions are also in **<malloc.h>**). They are not controlled by a compiler option, so you can use them in your program at any time.

## Which Should I Use?

Both sets of debugging functions have their benefits and drawbacks. Which you choose to use depends on your program, your problems, and your preference.

The debug memory management functions provide detailed information about all allocation requests you make with them in your program. You don't need to change any code to use the debug versions; you need only specify the **-qheapdebug** compiler option. However, because only calls that have been mapped to debug versions provide any information, you may have to rebuild many or all of your program's modules, which can be time-consuming.

On the other hand, the heap-checking functions perform more general checks on the heap at specific points in your program. You have greater control over where the checks occur. The heap-checking functions also provide compatibility with other compilers that offer these functions. You only have to rebuild the modules that contain the heap-checking calls. However, you have to change your source code to include these calls, which you will probably want to remove in your final code. Also, the heap-checking functions only tell you if the heap is consistent or not; they do not provide the details that the debug memory management functions do.

What you may choose to do is add calls to heap-checking functions in places you suspect possible memory problems. If the heap turns out to be corrupted, at that point you may want to rebuild with the **-qheapdebug** option.

**Note:** When the debug memory option **-qheapdebug** is specified, code is generated to *pre-initialize* the local variables for all functions. This makes it much more likely that uninitialized local variables will be found during the normal debug cycle rather than much later (usually when the code is optimized).

Regardless of which debugging functions you choose, your program requires additional memory to maintain internal information for these functions. If you are using fixed-size heaps, you may have to increase the heap size in order to use the debugging functions.

### Related Reading

- “Memory Management Functions” on page 179
- “Managing Memory with Multiple Heaps” on page 182
- “Types of Memory” on page 183
- “Creating and Using a Fixed Size Heap” on page 186
- “Creating and Using an Expandable Heap” on page 188
- “Debugging Programs with Heap Memory” on page 195
- “Changing the Default Heap Used in a Program”
- “Example of Creating and Using a User Heap” on page 190
- “Example of Creating and Using a Shared-Memory User Heap” on page 191
- “heapdebug” on page 270
- “\_debug\_calloc - Allocate and Initialize Memory” on page 407
- “\_debug\_free - Free Allocated Memory” on page 408
- “\_debug\_heapmin - Free Unused Memory in the Default Heap” on page 410
- “\_debug\_malloc - Allocate Memory” on page 412
- “\_debug\_memcpy - Copy Bytes” on page 413
- “\_debug\_memmove - Copy Bytes” on page 415
- “\_debug\_memset - Set Bytes to Value” on page 416
- “\_debug\_realloc - Reallocate Memory Block” on page 417
- “\_debug\_strcat - Concatenate Strings” on page 419
- “\_debug\_strcpy - Copy Strings” on page 421
- “\_debug\_strncat - Concatenate Strings” on page 422
- “\_debug\_strncpy - Copy Strings” on page 423
- “\_debug\_strnset - Set Characters in String” on page 425
- “\_debug\_strset - Set Characters in String” on page 426
- “\_debug\_ucalloc - Reserve and Initialize Memory from User Heap” on page 428
- “\_debug\_uheapmin - Free Unused Memory in User Heap” on page 430
- “\_debug\_umalloc - Reserve Memory Blocks from User Heap” on page 431
- “heapdebug” on page 270

## Changing the Default Heap Used in a Program

The regular memory management functions (`malloc` and so on) always use whatever heap is currently the default for that thread. The initial default heap for all C for AIX applications is the runtime heap provided by C for AIX. However, you can make your own heap the default by calling `_udfault`. Then all calls to the regular memory management functions allocate from your heap instead of the runtime heap.

The default heap changes only for the thread where you call `_udfault`. You can use a different default heap for each thread of your program if you choose.

This is useful when you want a component (such as a vendor library) to use a heap other than the C for AIX runtime heap, but you can't actually alter the source code to use heap-specific calls. For example, if you set the default heap to a shared heap then call a library function that calls `malloc`, the library allocates storage in shared memory.

Because `_udfault` returns the current default heap, you can save the return value and later use it to restore the default heap you replaced. You can also change the default back to the C for AIX runtime heap by calling `_udfault` and specifying `_RUNTIME_HEAP` (defined in `<umalloc.h>`). You can also use this macro with any of the heap-specific functions to explicitly allocate from the runtime heap.

## Related Reading

“Memory Management Functions” on page 179

“Managing Memory with Multiple Heaps” on page 182

“Types of Memory” on page 183

“Debugging Memory Heaps” on page 183

“Creating and Using a Fixed Size Heap”

“Creating and Using an Expandable Heap” on page 188

“Debugging Programs with Heap Memory” on page 195

“Example of Creating and Using a User Heap” on page 190

“Example of Creating and Using a Shared-Memory User Heap” on page 191

## Creating and Using a Fixed Size Heap

Before creating a heap, you must first allocate a block of memory large enough to hold the heap. The block must be large enough to satisfy all the memory requests your program will make of it, and also be able to hold internal information required to manage the heap. Once the block is fully allocated, further allocation requests to the heap will fail.

The internal information requires `_HEAP_MIN_SIZE` bytes (`_HEAP_MIN_SIZE` is defined in `<umalloc.h>`). You cannot create a heap smaller than this. Add the amount of memory your program requires to this value to determine the size of the block you need to get. Also make sure the block is the correct type (regular or shared) for the heap you are creating.

After you have allocated a block of memory, create the heap with `_ucreate`.

For example:

```
Heap_t fixedHeap; /* this is the "heap handle" */
/* get memory for internal info plus 5000 bytes for the heap */
static char block[_HEAP_MIN_SIZE + 5000];
fixedHeap = _ucreate(block, (_HEAP_MIN_SIZE+5000), /* block to use */
                    !_BLOCK_CLEAN, /* memory is not set to 0 */
                    _HEAP_REGULAR, /* regular memory */
                    NULL, NULL); /* we'll explain this later */
```

The `!_BLOCK_CLEAN` parameter indicates that the memory in the block has not been initialized to 0. If it were set to 0 (for example, by `memset`), you would specify `_BLOCK_CLEAN`. The `calloc` and `_ucalloc` functions use this information to improve their efficiency; if the memory is already initialized to 0, they don't need to initialize it.

The fourth parameter indicates what type of memory the heap contains: regular (`_HEAP_REGULAR`) or shared (`_HEAP_SHARED`). The different memory types are described in “Types of Memory” on page 183.

For a fixed-size heap, the last two parameters are always `NULL`.

## Using Your Heap

Once you have created your heap, you can open it for use by calling `_uopen`:

```
_uopen(fixedHeap);
```

This opens the heap for that particular process; if the heap is shared, each process that uses the heap needs its own call to `_uopen`.

You can then allocate and free from your own heap just as you would from the default heap. To allocate memory, use `_ucalloc` or `_umalloc`. These functions work just like `calloc` and `malloc`, except you specify the heap to use as well as the size of block that you want. For example, to allocate 1000 bytes from `fixedHeap`:

```
void *up;
up = _umalloc(fixedHeap, 1000);
```

To reallocate and free memory, use the regular `realloc` and `free` functions. Both of these functions always check what heap the memory came from, so you don't need to specify the heap to use. For example, the `realloc` and `free` calls in the following code fragment look exactly the same for both the default heap and your heap:

```
void *p, *up;
p = malloc(1000); /* allocate 1000 bytes from default heap */
up = _umalloc(fixedHeap, 1000); /* allocate 1000 from fixedHeap */
realloc(p, 2000); /* reallocate from default heap */
realloc(up, 100); /* reallocate from fixedHeap */
free(p); /* free memory back to default heap */
free(up); /* free memory back to fixedHeap */
```

For any object, you can find out what heap it was allocated from by calling `_mheap`. You can also get information about the heap itself by calling `_ustats`, which tells you:

- How much memory the heap holds (excluding memory used for overhead)
- How much memory is currently allocated from the heap
- What type of memory is in the heap
- The size of the largest contiguous piece of memory available from the heap

When you call any heap function, make sure the heap you specify is valid. If the heap is not valid, the behavior of the heap functions is undefined.

## Adding to a Fixed-Size Heap

Although you created the heap with a fixed size, you can add blocks of memory to it with `_uaddmem`. This can be useful if you have a large amount of memory that is allocated conditionally. Like the starting block, you must first allocate memory for a block of memory. This block will be added to the current heap, so make sure the block you add is the same type of memory as the heap you are adding it to.

For example, to add 64K to `fixedHeap`:

```
static char newblock[65536];
_uaddmem(fixedHeap, /* heap to add to */
         newblock, 65536, /* block to add */
         _BLOCK_CLEAN); /* sets memory to 0 */
```

Using `_uaddmem` is the only way to increase the size of a fixed heap.

**Note:** For every block of memory you add, a small number of bytes from it are used to store internal information. To reduce the total amount of overhead, it is better to add a few large blocks of memory than many small blocks.



## Destroying Your Heap

When you have finished using the heap, close it with `_uclose`. Once you have closed the heap in a process, that process can no longer allocate from or return memory to that heap. If other processes share the heap, they can still use it until you close it in each of them. Performing operations on a heap after you've closed it causes undefined behavior.

To finally destroy the heap, call `_udestroy`. If blocks of memory are still allocated somewhere, you can force the destruction. Destroying a heap removes it entirely even if it was shared by other processes. Again, performing operations on a heap after you've destroyed it causes undefined behavior.

After you destroy your fixed-size heap, it is up to you to return the memory for the heap (the initial block of memory you supplied to `_ucreate` and any other blocks added by `_uaddmem`) to the system.

### Related Reading

"Memory Management Functions" on page 179

"Managing Memory with Multiple Heaps" on page 182

"Types of Memory" on page 183

"Debugging Memory Heaps" on page 183

"Creating and Using an Expandable Heap"

"Debugging Programs with Heap Memory" on page 195

"Changing the Default Heap Used in a Program" on page 185

"Example of Creating and Using a User Heap" on page 190

"Example of Creating and Using a Shared-Memory User Heap" on page 191

## Creating and Using an Expandable Heap

When using a fixed-size heap, the initial block of memory must be large enough to satisfy all allocation requests made to it. You can also, however, create a heap that can expand and contract as your program needs demand.

With the C for AIX runtime heap, when not enough storage is available for your `malloc` request, the runtime gets additional storage from the system. Similarly, when you minimize the heap with `_heapmin` or when your program ends, the runtime returns the memory to the operating system.

When you create an expandable heap, you provide your own functions to do this work (we'll call them `getmore_fn` and `release_fn`, although you can name them whatever you choose). You specify pointers to these functions as the last two parameters to `_ucreate` (instead of the `NULL` pointers you used to create a fixed-size heap). For example:

```
Heap_t growHeap;
static char block[_HEAP_MIN_SIZE]; /* get block */
growHeap = _ucreate(block, _HEAP_MIN_SIZE, /* starting block */
                    !_BLOCK_CLEAN,      /* memory not set to 0 */
                    _HEAP_REGULAR,      /* regular memory */
                    getmore_fn,        /* function to expand heap */
                    release_fn);      /* function to shrink heap */
```

**Note:** You can use the same `getmore_fn` and `release_fn` for more than one heap, as long as the heaps use the same type of memory and your functions are not written specifically for one heap.



## Expanding Your Heap

When you call `_umalloc` (or a similar function) for your heap, `_umalloc` tries to allocate the memory from the initial block you provided to `_ucreate`. If not enough memory is there, it then calls your `getmore_fn`. Your `getmore_fn` then gets more memory from the operating system and adds it to the heap. It is up to you how you do this.

Your `getmore_fn` must have the following prototype:

```
void *(*getmore_fn)(Heap_t uh, size_t *size, int *clean);
```

The `uh` is the heap to be expanded.

The `size` is the size of the allocation request passed by `_umalloc`. You probably want to return enough memory at a time to satisfy several allocations; otherwise every subsequent allocation has to call `getmore_fn`, reducing your program's execution speed. Make sure that you update the `size` parameter. If you return more than the `size` requested.

Your function must also set the `clean` parameter to either `_BLOCK_CLEAN`, to indicate the memory has been set to 0, or `!_BLOCK_CLEAN`, to indicate that the memory has not been initialized.

The following fragment shows an example of a `getmore_fn`:

```
static void *getmore_fn(Heap_t uh, size_t *length, int *clean)
{
    char *newblock;
    /* round the size up to a multiple of 64K */
    *length = (*length / 65536) * 65536 + 65536;
    *clean = _BLOCK_CLEAN; /* mark the block as "clean" */
    return(newblock);      /* return new memory block */
}
```

Be sure that your `getmore_fn` allocates the right type of memory (regular or shared) for the heap. There are also special considerations for shared memory, as described under “Types of Memory” on page 183.

You can also use `_uaddmem` to add blocks to your heap, as you did for the fixed heap in “Creating and Using a Fixed Size Heap” on page 186. `_uaddmem` works exactly the same way for expandable heaps.

## Shrinking Your Heap

To coalesce the heap (return all blocks in the heap that are totally free to the system), use `_uheapmin`. `_uheapmin` works like `_heapmin`, except that you specify the heap to use.

When you call `_uheapmin` to coalesce the heap or `_udestroy` to destroy it, these functions call your `release_fn` to return the memory to the system. Again, it is up to you how you implement this function.

Your `release_fn` must have the following prototype:

```
void (*release_fn)(Heap_t uh, void *block, size_t size);
```

Where `uh` identifies the heap to be shrunk. The pointer `block` and its `size` are passed to your function by `_uheapmin` or `_udestroy`. Your function must return the memory pointed to by `block` to the system. For example:

```

static void release_fn(Heap_t uh, void *block, size_t size)
{
    free(block);
    return;
}

```

### Notes:

1. `_udestroy` calls your `release_fn` to return all memory added to the `uh` heap by your `getmore_fn` or by `_uaddmem`. However, you are responsible for returning the initial block of memory that you supplied to `_ucreate`.
2. Because a fixed-size heap has no `release_fn`, `_uheapmin` and `_udestroy` work slightly differently. Calling `_uheapmin` for a fixed-size heap has no effect but does not cause an error; `_uheapmin` simply returns 0. Calling `_udestroy` for a fixed-size heap marks the heap as destroyed, so no further operations can be performed on it, but returns no memory. It is up to you to return the heap's memory to the system.

### Related Reading

"Memory Management Functions" on page 179  
 "Managing Memory with Multiple Heaps" on page 182  
 "Types of Memory" on page 183  
 "Debugging Memory Heaps" on page 183  
 "Creating and Using a Fixed Size Heap" on page 186  
 "Debugging Programs with Heap Memory" on page 195  
 "Changing the Default Heap Used in a Program" on page 185  
 "Example of Creating and Using a User Heap"  
 "Example of Creating and Using a Shared-Memory User Heap" on page 191

## Example of Creating and Using a User Heap

The program below shows how you might create and use a heap.

Assuming that the program file is called `t.c`, compile it with the following command:

```
/usr/vac/bin/cc -qheapdebug t.c -lhu
```

```

#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>
static void *get_fn(Heap_t usrheap, size_t *length, int *clean)
{
    void *p;
    /* Round up to the next chunk size */
    *length = ((*length) / 65536) * 65536 + 65536;
    *clean = _BLOCK_CLEAN;
    p = calloc(*length,1);
    return (p);
}
static void release_fn(Heap_t usrheap, void *p, size_t size)
{
    free( p );
    return;
}
int main(void)
{
    void *initial_block;
    long rc;
    Heap_t myheap;
    char *ptr;
    int initial_sz;
    /* Get initial area to start heap */

```

```

initial_sz = 65536;
initial_block = malloc(initial_sz);
if(initial_block == NULL) return (1);
/* create a user heap */
myheap = _ucreate(initial_block, initial_sz, _BLOCK_CLEAN,
                 _HEAP_REGULAR, get_fn, release_fn);
if (myheap == NULL) return(2);
/* allocate from user heap and cause it to grow */
ptr = _umalloc(myheap, 100000);
_ufree(ptr);
/* destroy user heap */
if (_udestroy(myheap, _FORCE)) return(3);
/* return initial block used to create heap */
free(initial_block);
return 0;
}

```

### Related Reading

- “Memory Management Functions” on page 179
- “Managing Memory with Multiple Heaps” on page 182
- “Types of Memory” on page 183
- “Debugging Memory Heaps” on page 183
- “Creating and Using a Fixed Size Heap” on page 186
- “Creating and Using an Expandable Heap” on page 188
- “Debugging Programs with Heap Memory” on page 195
- “Changing the Default Heap Used in a Program” on page 185
- “Example of Creating and Using a Shared-Memory User Heap”

## Example of Creating and Using a Shared-Memory User Heap

The following program shows how you might implement a heap shared between a parent and several child processes.

Example of a User Heap - Parent Process (page 191) shows the parent process, which creates the shared heap. First the main program calls the `init` function to allocate shared memory from the operating system (using `CreateFileMapping`) and name the memory so that other processes can use it by name. The `init` function then creates and opens the heap. The loop in the main program performs operations on the heap, and also starts other processes. The program then calls the `term` function to close and destroy the heap.

Example of a Shared User Heap- Child Process (page 193) shows the process started by the loop in the parent process. This process uses `OpenFileMapping` to access the shared memory by name, then extracts the heap handle for the heap created by the parent process. The process then opens the heap, makes it the default heap, and performs some operations on it in the loop. After the loop, the process replaces the old default heap, closes the user heap, and ends.

### Example of a User Heap - Parent Process

```

/* The following program shows how you might implement
a heap shared between a parent and several child processes.
Example of a Shared User Heap - Parent Process shows the parent
process, which creates the shared heap. First
the main program calls the init function to allocate shared memory from the operating system (using
CreateFileMapping) and name the memory so that other processes can use it by name. The init function
then creates and opens the heap. The loop in the main program performs operations on the heap, and also
starts other processes. The program then calls the term function to close and destroy the heap.
*/
#include <umalloc.h>
#include <stdio.h>
#include <stdlib.h>

```

```

#include <string.h>
#define PAGING_FILE 0xFFFFFFFF
#define MEMORY_SIZE 65536
#define BASE_MEM (VOID*)0x01000000
static HANDLE hFile; /* Handle to memory file */
static void* hMap; /* Handle to allocated memory */
typedef struct mem_info {
    void * pBase;
    Heap_t pHeap;
} MEM_INFO_T;
/*-----*/
/* inithp: */
/* Function to create and open the heap with a named shared memory object */
/*-----*/
static Heap_t inithp(size_t heap_size)
{
    MEM_INFO_T info; /* Info structure */
    /* Allocate shared memory from the system by creating a shared memory */
    /* pool basing it out of the system paging (swapper) file. */
    hFile = CreateFileMapping( (HANDLE) PAGING_FILE,
        NULL,
        PAGE_READWRITE,
        0,
        heap_size + sizeof(Heap_t),
        "MYNAME_SHAREMEM" );

    if (hFile == NULL) {
        return NULL;
    }
    /* Map the file to this process' address space, starting at an address */
    /* that should also be available in child processe(s) */
    hMap = MapViewOfFileEx( hFile, FILE_MAP_WRITE, 0, 0, 0, BASE_MEM );
    info.pBase = hMap;
    if (info.pBase == NULL) {
        return NULL;
    }
    /* Create a fixed sized heap. Put the heap handle as well as the */
    /* base heap address at the beginning of the shared memory. */
    info.pHeap = _ucreate((char *)info.pBase + sizeof(info),
        heap_size - sizeof(info),
        !_BLOCK_CLEAN,
        _HEAP_SHARED | _HEAP_REGULAR,
        NULL, NULL);

    if (info.pBase == NULL) {
        return NULL;
    }
    memcpy(info.pBase, info, sizeof(info));
    if (_uopen(info.pHeap) { /* Open heap and check result */
        return NULL;
    }
    return info.pHeap;
}
/*-----*/
/* termhp: */
/* Function to close and destroy the heap */
/*-----*/
static int termhp(Heap_t uheap)
{
    if (_uclose(uheap)) /* close heap */
        return 1;
    if (_udestroy(uheap, _FORCE)) /* force destruction of heap */
        return 1;
    UnmapViewOfFile(hMap); /* return memory to system */
    CloseHandle(hFile);
    return 0;
}
/*-----*/
/* main: */

```

```

/* Main function to test creating, writing to and destroying a shared heap.
/* -----*/
int main(void)
{
    int i, rc;                /* Index and return code */
    Heap_t uheap;            /* heap to create */
    void *init_block;        /* initial block to use */
    char *p;                 /* for allocating from heap */
    /* call init function to create and open the heap
    /*
    uheap = inithp(MEMORY_SIZE);
    if (uheap == NULL)        /* check for success */
        return 1;           /* if failure, return non zero */
    /* perform operations on uheap
    /*
    for (i = 1; i <= 5; i++)
    {
        p = _umalloc(uheap, 10); /* allocate from uheap */
        if (p == NULL)
            return 1;
        memset(p, 'M', _msize(p)); /* set all bytes in p to 'M' */
        p = realloc(p,50);        /* reallocate from uheap */
        if (p == NULL)
            return 1;
        memset(p, 'R', _msize(p)); /* set all bytes in p to 'R' */
    }
    /* Start a second process which accesses the heap
    /*
    if (system("memshr2.exe"))
        return 1;
    /* Take a look at the memory that we just wrote to. Note that memshr.c
    /* and memshr2.c should have been compiled specifying the /Tm+ flag.
    /*
#ifdef DEBUG
    _udump_allocated(uheap, -1);
#endif
    /* call term function to close and destroy the heap
    /*
    rc = termhp(uheap);
#ifdef DEBUG
    printf("memshr ending... rc = %d\n", rc);
#endif
    return rc;
}

```

### Example of a Shared User Heap - Child Process

```

/* Example of a Shared User Heap - Child Process shows
the process started by the loop in the parent process.
This process uses OpenFileMapping to access the shared memory
by name, then extracts the heap handle for the heap created
by the parent process. The process then opens the heap,
makes it the default heap, and performs some operations
on it in the loop. After the loop, the process replaces
the old default heap, closes the user heap, and ends.
*/
#include <umalloc.h>
#include <stdio.h>
#include <stdlib.h>

```

```

#include <string.h>
static HANDLE hFile;          /* Handle to memory file          */
static void* hMap;           /* Handle to allocated memory     */
typedef struct mem_info {
    void * pBase;
    Heap_t pHeap;
} MEM_INFO_T;
/*-----*/
/* inithp: Subprocess Version          */
/* Function to create and open the heap with a named shared memory object */
/*-----*/
static Heap_t inithp(void)
{
    MEM_INFO_T info;          /* Info structure          */
    /* Open the shared memory file by name. The file is based on the
    /* system paging (swapper) file.          */
    hFile = OpenFileMapping(FILE_MAP_WRITE, FALSE, "MYNAME_SHAREMEM");
    if (hFile == NULL) {
        return NULL;
    }
    /* Figure out where to map this file by looking at the address in the
    /* shared memory where the memory was mapped in the parent process.
    hMap = MapViewOfFile( hFile, FILE_MAP_WRITE, 0, 0, sizeof(info) );
    if (hMap == NULL) {
        return NULL;
    }
    /* Extract the heap and base memory address from shared memory
    memcpy(info, hMap, sizeof(info));
    UnmapViewOfFile(hMap);
    hMap = MapViewOfFileEx( hFile, FILE_MAP_WRITE, 0, 0, 0, info.pBase );
    if (_uopen(info.pHeap)) {          /* Open heap and check result
        return NULL;
    }
    return info.pHeap;
}
/*-----*/
/* termhp:                             */
/* Function to close my view of the heap */
/*-----*/
static int termhp(Heap_t uheap)
{
    if (_uclose(uheap))                /* close heap          */
        return 1;
    UnmapViewOfFile(hMap);              /* return memory to system
    CloseHandle(hFile);
    return 0;
}
/*-----*/
/* main:                                 */
/* Main function to test creating, writing to and destroying a shared
/* heap.
/*-----*/
int main(void)
{
    int rc, i;                        /* for return code, loop iteration
    Heap_t uheap, oldheap;             /* heap to create, old default heap
    char *p;                           /* for allocating from the heap
    /*
    /* Get the heap storage from the shared memory
    /*
    uheap = inithp();
    if (uheap == NULL)
        return 1;
    /*
    /* Register uheap as default runtime heap, save old default
    /*
    oldheap = _udefault(uheap);

```

```

    if (oldheap == NULL) {
        return termhp(uheap);
    }
    /*
    /* Perform operations on uheap
    /*
    for (i = 1; i <= 5; i++)
    {
        p = malloc(10);    /* malloc uses default heap, which is now uheap*/
        memset(p, 'M', _msize(p));
    }
    /*
    /* Replace original default heap and check result
    /*
    /*
    if (uheap != _default(oldheap)) {
        return termhp(uheap);
    }
    /*
    /* Close my views of the heap
    /*
    /*
    rc = termhp(uheap);
    #ifdef DEBUG
        printf("Returning from memshr2 rc = %d\n", rc);
    #endif
    return rc;
}

```

### Related Reading

- “Memory Management Functions” on page 179
- “Managing Memory with Multiple Heaps” on page 182
- “Types of Memory” on page 183
- “Debugging Memory Heaps” on page 183
- “Creating and Using a Fixed Size Heap” on page 186
- “Creating and Using an Expandable Heap” on page 188
- “Debugging Programs with Heap Memory”
- “Changing the Default Heap Used in a Program” on page 185
- “Example of Creating and Using a User Heap” on page 190

## Debugging Programs with Heap Memory

C for AIX provides debug versions of both general memory management functions and heap-specific memory management functions. To automatically call the debug versions of these functions, specify the **-qheapdebug** compiler option when compiling your program. Bear in mind that specifying this option can significantly increase the memory requirements and running time of your program.

### Memory Allocation Fill Pattern

Some debug functions set all the memory they allocate to a specified fill pattern. This lets you easily locate areas in memory that your program uses.

The **debug\_malloc**, **debug\_realloc**, and **debug\_umalloc** functions sets allocated memory to a default repeating 0xAA fill pattern. To enable this fill pattern, export the HD\_FILL environment variable.

The **debug\_free** function sets all free memory to a repeating 0xFB fill pattern.

## Skipping Heap Checks

Each debug function calls `_heap_check` (or `_uheap_check`) to check the heap. Although this is useful, it can also increase your program's memory requirements and decrease its execution speed.

To reduce the overhead of checking the heap on every debug memory management function, you can control how often the functions check the heap with the `HD_SKIP` environment variable. You will not need to do this for most of your applications unless the application is extremely memory intensive.

Set `HD_SKIP` like any other environment variable. The syntax for `HD_SKIP` is:

```
set HD_SKIP=increment, [start]
```

where:

*increment*            Specifies how often you want the debug functions to check the heap.  
*start*                 Optional. Use this parameter to start skipping heap checks after *start* calls to debug functions.

**Note:** The comma separating the parameters is optional.

When you use the *start* parameter to start skipping heap checks, you are trading off heap checks that are done implicitly against program execution speed. You should therefore start with a small increment (like 5) and slowly increase until the application is usable.

For example, if you specify:

```
set HD_SKIP=10
```

then every tenth debug memory function call performs a heap check. If you specify:

```
set HD_SKIP=5,100
```

then after 100 debug memory function calls, only every fifth call performs a heap check. Other than the heap check, the debug functions behave exactly the same as usual.

## Using Stack Traces

Stack contents are traced for each allocated memory object. If the contents of an object's stack change, the traced contents are dumped.

The trace size is controlled by the `HD_STACK` environment variable. If this variable is not set, the compiler assumes a stack size of 10. To disable stack tracing, set the `HD_STACK` environment variable to 0.



### Related Reading

“Memory Management Functions” on page 179  
“Managing Memory with Multiple Heaps” on page 182  
“Types of Memory” on page 183  
“Debugging Memory Heaps” on page 183  
“Creating and Using a Fixed Size Heap” on page 186  
“Creating and Using an Expandable Heap” on page 188  
“Changing the Default Heap Used in a Program” on page 185  
“Example of Creating and Using a User Heap” on page 190  
“Example of Creating and Using a Shared-Memory User Heap” on page 191  
“\_debug\_calloc - Allocate and Initialize Memory” on page 407  
“\_debug\_free - Free Allocated Memory” on page 408  
“\_debug\_heapmin - Free Unused Memory in the Default Heap” on page 410  
“\_debug\_malloc - Allocate Memory” on page 412  
“\_debug\_memcpy - Copy Bytes” on page 413  
“\_debug\_memmove - Copy Bytes” on page 415  
“\_debug\_memset - Set Bytes to Value” on page 416  
“\_debug\_realloc - Reallocate Memory Block” on page 417  
“\_debug\_strcat - Concatenate Strings” on page 419  
“\_debug\_strcpy - Copy Strings” on page 421  
“\_debug\_strncat - Concatenate Strings” on page 422  
“\_debug\_strncpy - Copy Strings” on page 423  
“\_debug\_strnset - Set Characters in String” on page 425  
“\_debug\_strset - Set Characters in String” on page 426  
“\_debug\_ucalloc - Reserve and Initialize Memory from User Heap” on page 428  
“\_debug\_uheapmin - Free Unused Memory in User Heap” on page 430  
“\_debug\_umalloc - Reserve Memory Blocks from User Heap” on page 431  
“heapdebug” on page 270

---

## Writing Optimized Program Source Code

This page contains tips for writing code to take advantage of the optimization features of the compiler. The following language elements are discussed:

- “Variables”
- “Pointers” on page 198
- “Functions” on page 199
- “Function Arguments” on page 199
- “Expressions” on page 199
- “Critical Loops” on page 200
- “Conversions” on page 201
- “Arithmetic Constructions” on page 201
- “Using Inlined Components” on page 202

You can also refer to the *Optimization Guide for Fortran, C, and C++* for more information about optimizing and tuning your code.

### Related Reading

“Program Optimization with the C for AIX Compiler” on page 23

## Variables

Use local variables, preferably automatic variables, as much as possible. The compiler can accurately analyze the use of local variables, but it has to make several worst-case assumptions about global

variables. These assumptions tend to hinder optimization. For example, if you write a function that uses external variables heavily, and that function also calls several external functions, the compiler assumes that every call to an external function could change the value of every external variable. If you know that none of the function calls affects the global variables that you are using, and you have to read them frequently with function calls interspersed, copy the global variables to local variables and then use these local variables. The compiler can then perform optimization that it could not otherwise perform.

If you must use global variables, use static variables with file scope rather than external variables wherever possible. In a file with several related functions and static variables, the optimizer can gather and use more information about how the variables are affected.

To access an external variable, the compiler has to make an extra memory access to obtain the address of the variable. When the compiler removes extraneous address loads, it has to use a register to keep the address. Using many external variables simultaneously takes up many registers. Those that cannot fit into registers during optimization are spilled into memory. Because all elements of an external structure use the same base address, you should group external data into structures or arrays wherever it makes sense to do so.

The “**#pragma isolated\_call Preprocessor Directive**” on page 371 preprocessor directive can improve the runtime performance of optimized code by allowing the compiler to make less pessimistic assumptions about the storage of external and static variables.

Because the compiler treats register variables the same as it does automatic variables, you do not gain anything by declaring register variables. Note that this differs from other implementations, where using the register attribute can greatly affect program performance.

#### **Related Reading**

“Program Optimization with the C for AIX Compiler” on page 23

“Writing Optimized Program Source Code” on page 197

“Pointers”

“Functions” on page 199

“Function Arguments” on page 199

“Expressions” on page 199

“Critical Loops” on page 200

“Conversions” on page 201

“Arithmetic Constructions” on page 201

“Using Inlined Components” on page 202

## **Pointers**

Keeping track of pointers during optimization is difficult and in some cases impossible. Using pointers inhibits most memory optimization (such as dead store elimination and store motion).

Using the “**#pragma disjoint Preprocessor Directive**” on page 366 preprocessor directive to list identifiers that do not share the same physical storage can improve the runtime performance of optimized code.

Also see “**assert**” on page 238 for information on applying aliasing assertions to pointers in your compilation unit.

### Related Reading

“Program Optimization with the C for AIX Compiler” on page 23  
“Writing Optimized Program Source Code” on page 197  
“Variables” on page 197  
“Functions”  
“Function Arguments”  
“Expressions”  
“Critical Loops” on page 200  
“Conversions” on page 201  
“Arithmetic Constructions” on page 201  
“Using Inlined Components” on page 202

## Functions

Declare nonmember functions as **static** whenever possible. This will speed up calls to the function.

### Related Reading

“Program Optimization with the C for AIX Compiler” on page 23  
“Writing Optimized Program Source Code” on page 197  
“Variables” on page 197  
“Pointers” on page 198  
“Function Arguments”  
“Expressions”  
“Critical Loops” on page 200  
“Conversions” on page 201  
“Arithmetic Constructions” on page 201  
“Using Inlined Components” on page 202

## Function Arguments

Optimization is effective when function arguments are used. It is usually better to pass a value as an argument to a function than to let the function take the value from a global variable.

The “**#pragma isolated\_call Preprocessor Directive**” on page 371 preprocessor directive lists functions that have no side effects. Using the pragma to list functions that do not have side effects, that is, that do not modify global storage, can improve the runtime performance of optimized code.

### Related Reading

“Program Optimization with the C for AIX Compiler” on page 23  
“Writing Optimized Program Source Code” on page 197  
“Variables” on page 197  
“Pointers” on page 198  
“Functions”  
“Expressions”  
“Critical Loops” on page 200  
“Conversions” on page 201  
“Arithmetic Constructions” on page 201  
“Using Inlined Components” on page 202

## Expressions

If components of an expression are duplicate expressions, code them either at the left end of the expression or within parentheses. For example:

```
a = b*(x*y*z);      /* Duplicates recognized */  
c = x*y*z*d;  
e = f + (x + y);
```

```

g = x + y + h;
a = b*x*y*z;      /* No duplicates recognized */
c = x*y*z*d;
e = f + x + y;
g = x + y + h;

```

When components of an expression in a loop are constant, code the expressions either at the left end of the expression, or within parentheses. If **c**, **d**, and **e** are constant and **v**, **w**, and **x** are variable, the following examples show the difference in evaluation:

```

v*w*x*(c*d*e);      /* Loop invariant expressions recognized */
c + d + e + v + w + x;
v*w*x*c*d*e;        /* Optimization required for loop invariant */
v + w + x + c + d + e; /*      expressions to be recognized      */

```

For integer expressions, the loop invariant expression will be recognized if **-O** is specified. For floating-point expressions, the loop invariant expression will be recognized if **-O3** is specified.

### Related Reading

- “Program Optimization with the C for AIX Compiler” on page 23
- “Writing Optimized Program Source Code” on page 197
- “Variables” on page 197
- “Pointers” on page 198
- “Functions” on page 199
- “Function Arguments” on page 199
- “Critical Loops”
- “Conversions” on page 201
- “Arithmetic Constructions” on page 201
- “Using Inlined Components” on page 202
- “**O**, optimize” on page 302

## Critical Loops

If your program contains a short, heavily referenced **for** loop, consider expanding the code to a straight sequence of statements. For example:

```

array[0] = b[k+1]*c[m+1];
array[1] = b[k+2]*c[m+2];
array[2] = b[k+3]*c[m+3];
array[3] = b[k+4]*c[m+4];
array[4] = b[k+5]*c[m+5];

```

would run faster than:

```

for (i = 0; i < 5; i++)
    array[i] = b[k+i]*c[m+i];

```

The compiler will perform automatic unrolling of inner loops when the **-O3** option is specified. In this case, the compiler will unroll the loop once.

### Related Reading

“Program Optimization with the C for AIX Compiler” on page 23  
“Writing Optimized Program Source Code” on page 197  
“Variables” on page 197  
“Pointers” on page 198  
“Functions” on page 199  
“Function Arguments” on page 199  
“Expressions” on page 199  
“Conversions”  
“Arithmetic Constructions”  
“Using Inlined Components” on page 202  
**“O, optimize” on page 302**

## Conversions

Avoid forcing the compiler to convert numbers between integer and floating-point internal representations. Conversions require several instructions, including some double-precision floating-point arithmetic. For example:

```
float array[10];
float x = 1.0;
int i;
for (i = 0; i < 9; i++) {      /* No conversions needed */
    array[i] = array[i]*x;
    x = x + 1.0;
}
for (i = 0; i < 9; i++)      /* Multiple conversions needed */
    array[i] = array[i]*i;
```

When you must use mixed-mode arithmetic, code the fixed-point and floating-point arithmetic in separate computations wherever possible.

### Related Reading

“Program Optimization with the C for AIX Compiler” on page 23  
“Writing Optimized Program Source Code” on page 197  
“Variables” on page 197  
“Pointers” on page 198  
“Functions” on page 199  
“Function Arguments” on page 199  
“Expressions” on page 199  
“Critical Loops” on page 200  
“Arithmetic Constructions”  
“Using Inlined Components” on page 202

## Arithmetic Constructions

Wherever possible, use multiplication rather than division. For example:

```
x*(1.0/3.0);
```

produces faster code than:

```
x/3.0;
```

Assigning the reciprocal of the divisor to a temporary variable and then multiplying by that variable is beneficial, especially if you divide many values by the same number in your code. This is attempted by the compiler when the **-O3** option is specified.

## Related Reading

“Program Optimization with the C for AIX Compiler” on page 23

“Writing Optimized Program Source Code” on page 197

“Variables” on page 197

“Pointers” on page 198

“Functions” on page 199

“Function Arguments” on page 199

“Expressions” on page 199

“Critical Loops” on page 200

“Conversions” on page 201

“Using Inlined Components”

“O, optimize” on page 302

## Using Inlined Components

By default, the compiler inlines certain library functions, meaning that it replaces the function call with the actual code for the function at the point where the call was made. These library functions are called intrinsic or built-in functions.

You can also request that the compiler inline the code for your own functions. There are benefits and drawbacks of, and restrictions on, inlining user code.

There are two ways to inline user code:

1. Use the C for AIX `_inline`, `_Inline`, and `__inline` keywords to specify which functions you want to have inlined. You must specify the `Q` or `-qinline` options to turn inlining on.
2. Use the `-Q` or `-qinline` option with a value parameter to automatically inline functions smaller than the value specified.

You should use inlining only for very small functions. See `-Q` or `-qinline` for more information about the inlining option.

**Note:** Requesting that a function be inlined makes it a candidate for inlining but does not necessarily mean that the function will be inlined. In all cases, the compiler ultimately decides whether a function is inlined.

## Benefits of Inlining

Inlining user code eliminates the overhead of the function call and linkage, and also exposes the function's code to the optimizer, resulting in faster code performance. Inlining produces the best results when:

- The overhead for the function is significant; for example, when functions are called within nested loops.
- The inlined function provides additional opportunities for optimization, such as when constant arguments are used.

For example, given the following function:

```
void glen(int a, int b)
{
    if (a == 10)
    {
        switch(b)
        {
            case 1 : .
                :
            case 20: puts("b is 20");
                    break;
            case 30: .
                :
        }
    }
}
```

```

        default: .
            :
        }
    }
}

```

and assuming your program calls `glen` several times with constant arguments, for example, `glen(10, 20);`, each call to `glen` causes the `if` and `switch` expressions to be evaluated. If `glen` is inlined, the compiler can then optimize the function. The evaluation of the `if` and `switch` statements can be done at compile time, and the function code can then be reduced to only the `puts` statement from case 20.

The best candidates for inlining are small functions that are called often. Use the Performance Analyzer or a profiler to determine which functions to inline to obtain the best results.

## Drawbacks of Inlining

Inlining user code usually results in a larger executable module because the code for the function is included at each call site. Because of the extra optimizations that can be performed, the difference in size may be less than the size of the function multiplied by the number of calls.

Inlining can also result in slower program performance, especially if you use auto-inlining. Because auto-inlining looks only at the number of ACUs for a function, the functions that are inlined are not always the best candidates for inlining. As much as possible, use the `_Inline` or `inline` keyword to choose the functions to be inlined.

When you use inlining, you need more stack space. When a function is called, its local storage is allocated at the time of the call and freed when it returns to the calling function. If that same function is inlined, its storage is allocated when the function that calls it is entered, and is not freed until that calling function ends. Ensure that you have enough stack space for the local storage of the inlined functions, in order to avoid a stack overflow.

## Restrictions on Inlining

The following restrictions apply to inlining:

If the definition and reference to a given function reside in different files, all such files must be compiled and linked using the `-qipa` compiler option. To inline across source files, you must place the function definition (qualified with `_Inline`) in a header file that is included by all source files where the function is to be inlined.

Turn off inlining if you plan to debug your executable module. Inlining can make debugging difficult. For example, if you set an entry breakpoint for a function call but the function is inlined, the breakpoint will not work.

The Performance Analyzer treats an inlined function as part of the function in which it is inlined.

A function is not inlined during an inline expansion of itself. For a function that is directly recursive, the call to the function from within itself is not inlined. For example, given three functions to be inlined, A, B, and C, where:

1. A calls B
2. B calls C
3. C calls back to B

the following inlining takes place:

1. The call to B from A is inlined.
2. The call to C from B is inlined.
3. The call to B from C is not inlined because it is made from within an inline expansion of B itself.

### Related Reading

“Program Optimization with the C for AIX Compiler” on page 23

“Writing Optimized Program Source Code” on page 197

“Variables” on page 197

“Pointers” on page 198

“Functions” on page 199

“Function Arguments” on page 199

“Expressions” on page 199

“Critical Loops” on page 200

“Conversions” on page 201

“Arithmetic Constructions” on page 201

“\_Inline, \_inline, \_\_inline”

“Q” on page 314

## **`_Inline, _inline, __inline`**

C for AIX provides keywords that you can use to specify functions that you want the compiler to 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 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. If you turn optimization on (`/O+`), `/Oi+` becomes the default.

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 fragment 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; }
```

**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.



**Related Reading**

“Program Optimization with the C for AIX Compiler” on page 23

“Using Inlined Components” on page 202

“Writing Optimized Program Source Code” on page 197

“inline” on page 277

“Q” on page 314



---

## Chapter 8. Using C for AIX with Other Programming Languages

With the C for AIX compiler, you can call functions written in other XL languages from your C program. Similarly, the other XL language programs can call functions written in C for AIX. This and related pages give you information about how to use interlanguage calls in your C program. You should already be familiar with the syntax of the languages you are using.

### Related Reading

"Interlanguage Calling Conventions"

"Corresponding Data Types"

"Using the Subroutine Linkage Conventions in Interlanguage Calls" on page 209

"Sample Program: C Calling Fortran" on page 214

---

## Interlanguage Calling Conventions

You should follow these recommendations when writing C for AIX 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. Both fold external identifiers to lowercase (by default).
- Avoid using the underscore (\_) and dollar sign (\$) as the first character in identifiers, to prevent conflict with the naming conventions for the C language library.
- Avoid using long identifier names. The maximum number of significant characters in identifiers is 250 characters.

### Related Reading

"Corresponding Data Types"

"Using the Subroutine Linkage Conventions in Interlanguage Calls" on page 209

"Sample Program: C Calling Fortran" on page 214

---

## Corresponding Data Types

The following table shows the correspondence between the data types available in C for AIX, C Set ++ for AIX, 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.

| Correspondence of Data Types among C, C++, Fortran, and Pascal |                    |                      |
|----------------------------------------------------------------|--------------------|----------------------|
| C and C++ Data Types                                           | Fortran Data Types | Pascal Data Types    |
| char                                                           | CHARACTER          | CHAR                 |
| signed char                                                    | INTEGER*1<br>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              |
| unsigned long int                                              | LOGICAL*4          | —                    |
| signed long long int                                           | INTEGER*8          | —                    |

|                                                             |                                   |                            |
|-------------------------------------------------------------|-----------------------------------|----------------------------|
| unsigned long long int                                      | LOGICAL*8                         | —                          |
| float                                                       | REAL<br>REAL*4                    | SHORTREAL                  |
| double                                                      | REAL*8<br>DOUBLE PRECISION        | REAL                       |
| long double                                                 | REAL*8<br>DOUBLE PRECISION        | REAL                       |
| long double (with <b>-qlongdouble</b> or <b>-qldbl128</b> ) | REAL*16                           | —                          |
| structure of two floats                                     | COMPLEX<br>COMPLEX*4              | RECORD of two SHORTREALS   |
| structure of two doubles                                    | COMPLEX*16<br>DOUBLE COMPLEX      | RECORD of two REALS        |
| structure of two long doubles                               | 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 <b>-qalign=pack</b> )                       | 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**, **strncpm**, and **strncpy** functions of the C runtime library. These functions are described in the *AIX Version 4 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 Reading

- “Interlanguage Calling Conventions” on page 207
- “Using the Subroutine Linkage Conventions in Interlanguage Calls”
- “Sample Program: C Calling Fortran” on page 214
- “macpstr” on page 295

---

## Using the Subroutine Linkage Conventions in Interlanguage Calls

The *subroutine linkage conventions* describes 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. The *AIX Version 4 Assembler Language Referencedescribes the Subroutine Linkage Convention* in detail.

The RISC System/6000 linkage convention provides fast and efficient subroutine linkage between languages. It specifies how parameters are passed, taking full advantage of the large number of floating-point registers (FPRs) and general-purpose registers (GPRs), and minimizes the saving and restoring of registers on subroutine entry and exit.

- “Interlanguage Calls - Parameter Passing” on page 210
- “Interlanguage Calls - Call by Reference Parameters” on page 210
- “Interlanguage Calls - Call by Value Parameters” on page 211
- “Interlanguage Calls - Rules for Passing Parameters by Value” on page 211
- “Interlanguage Calls - Pointers to Functions” on page 212
- “Interlanguage Calls - Function Return Values” on page 213
- “Interlanguage Calls - Stack Floor” on page 213
- “Interlanguage Calls - Stack Overflow” on page 213
- “Interlanguage Calls - Traceback Table” on page 214
- “Interlanguage Calls - Type Encoding and Checking” on page 214

#### Related Reading

- “Interlanguage Calling Conventions” on page 207
- “Corresponding Data Types” on page 207
- “Sample Program: C Calling Fortran” on page 214

## 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 Reading

"Interlanguage Calling Conventions" on page 207

"Corresponding Data Types" on page 207

"Using the Subroutine Linkage Conventions in Interlanguage Calls" on page 209

"Sample Program: C Calling Fortran" on page 214

## 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 function 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 Reading

“Interlanguage Calling Conventions” on page 207

“Corresponding Data Types” on page 207

“Using the Subroutine Linkage Conventions in Interlanguage Calls” on page 209

“Sample Program: C Calling Fortran” on page 214

## 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 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  $(\text{sizeof}(\text{struct } X)+3)/4$  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” on page 212 for more information.

### Related Reading

“Interlanguage Calling Conventions” on page 207

“Corresponding Data Types” on page 207

“Using the Subroutine Linkage Conventions in Interlanguage Calls” on page 209

“Interlanguage Calls - Pointers to Functions” on page 212

“Sample Program: C Calling Fortran” on page 214

“ldbl128, longdouble” on page 289

## Interlanguage Calls - Rules for Passing Parameters by Value

The following is an example of a call to a prototyped function:

```
int i, j, k;
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:

| Will be passed in:  | Storage mapping of the PARM area on the Stack |                                                  |
|---------------------|-----------------------------------------------|--------------------------------------------------|
| R3                  | 0                                             | i                                                |
| R4                  | 4                                             | j                                                |
| R5                  | 8                                             | k                                                |
| FP1 (R6, R7 unused) | 12                                            | d1                                               |
|                     | 16                                            |                                                  |
| FP2 (R8 unused)     | 20                                            | f1                                               |
| R9                  | 24                                            | //// //// ////  c                                |
|                     |                                               | ← right justified for a nonprototyped C function |
| FP3 (R10 unused)    | 28                                            | d2                                               |
|                     | 32                                            |                                                  |
| Stack               | 36                                            | //// //// ////  s1                               |
|                     |                                               | ← right justified for a nonprototyped C function |

### 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 ( $P_1 \dots P_n$ ) must be large enough to hold the largest parameter list.

### Related Reading

- “Interlanguage Calling Conventions” on page 207
- “Corresponding Data Types” on page 207
- “Using the Subroutine Linkage Conventions in Interlanguage Calls” on page 209
- “Sample Program: C Calling Fortran” on page 214

## 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 Reading

“Interlanguage Calling Conventions” on page 207

“Corresponding Data Types” on page 207

“Using the Subroutine Linkage Conventions in Interlanguage Calls” on page 209

“Sample Program: C Calling Fortran” on page 214

## 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.
- **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 Reading

“Interlanguage Calling Conventions” on page 207

“Corresponding Data Types” on page 207

“Using the Subroutine Linkage Conventions in Interlanguage Calls” on page 209

“Sample Program: C Calling Fortran” on page 214

## Interlanguage Calls - Stack Floor

The *stack floor* is a system-defined address below which the stack cannot grow. All programs in the system must avoid accessing locations in the stack segment that are below the stack floor.

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 Reading

“Interlanguage Calling Conventions” on page 207

“Corresponding Data Types” on page 207

“Using the Subroutine Linkage Conventions in Interlanguage Calls” on page 209

“Sample Program: C Calling Fortran” on page 214

## 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 Reading

“Interlanguage Calling Conventions” on page 207

“Corresponding Data Types” on page 207

“Using the Subroutine Linkage Conventions in Interlanguage Calls” on page 209

“Sample Program: C Calling Fortran” on page 214

## Interlanguage Calls - Traceback Table

The compiler supports the traceback mechanism, which is required by the AIX Version 4 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 Reading

“Interlanguage Calling Conventions” on page 207

“Corresponding Data Types” on page 207

“Using the Subroutine Linkage Conventions in Interlanguage Calls” on page 209

“Sample Program: C Calling Fortran”

## Interlanguage Calls - Type Encoding and Checking

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

The C for AIX compiler uses a scheme for early detection that encodes information about all external symbols (data and programs). If the “**extchk**” on **page 258** option has been specified, this information about external symbols is checked at bind or load time for consistency.

The *Assembler Language Reference for the AIX RISC System/6000* 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 Reading

“Interlanguage Calling Conventions” on page 207

“Corresponding Data Types” on page 207

“Using the Subroutine Linkage Conventions in Interlanguage Calls” on page 209

“Sample Program: C Calling Fortran”

---

## 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 Reading**

"Interlanguage Calling Conventions" on page 207

"Corresponding Data Types" on page 207

"Using the Subroutine Linkage Conventions in Interlanguage Calls" on page 209



---

## Appendix A. Compiler Options

The compiler options 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.

Uppercase letters in the option, suboption, or **#pragma options** keyword syntax represent its valid abbreviation. For example, both of the following are acceptable specifications of the **LANGlvl** option in a source file:

```
#pragma options lang=ansi
#pragma options langlvl=ansi
```

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

### Related Reading

- “Invoking the Compiler” on page 8
- “Specifying Compiler Options on the Command Line” on page 10
- “Specifying Compiler Options in Your Program Source Files” on page 12
- “Specifying Compiler Options in a Configuration File” on page 13
- “Resolving Conflicting Compiler Options”
- “Compiler Options and Their Defaults” on page 218
- “Options that Specify Compiler Characteristics” on page 226
- “Options that Specify Debugging Features” on page 227
- “Options that Specify Preprocessor Options” on page 228
- “Options that Specify Compiler Output” on page 228
- “Options that Specify the Compiler Object Code Produced” on page 229
- “Options that Specify Linkage Options” on page 230

---

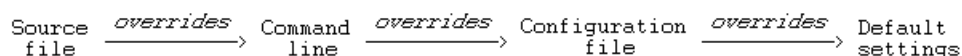
## Resolving Conflicting Compiler Options

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 **-ldirectory** 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                |
| attr          | -qattr=FULL                             | -qattr=FULL                |
| -p            | -pg                                     | Last option specified      |
| -qhsflt       | -qrndsngl -qspnans                      | -qhsflt                    |
| -qhssngl      | -qrndsngl -qspnans                      | hssngl                     |
| -E            | -P -o -S                                | -E                         |
| -P            | -c -o -S                                | -P                         |
| -#            | -v                                      | -#                         |
| -F            | -B -t -W configuration file settings    | -B -t -W                   |
| -S            | -c                                      | -S                         |

#### Related Reading

“Specifying Compiler Options for Architecture-Specific, 32- or 64-bit Compilation” on page 14

“Invoking the Compiler” on page 8

“Specifying Compiler Options on the Command Line” on page 10

“Specifying Compiler Options in Your Program Source Files” on page 12

“Specifying Compiler Options in a Configuration File” on page 13

---

## Compiler Options and Their Defaults

This page lists all C for AIX compiler options, specifying each option’s type and if it exists, default value. Where a \* appears beside the default value for a compiler option, see the description for that option for special notes regarding the default value.

To get detailed information on any option listed, click on the that option’s name in the table.

| Option Name             | Type         | Default                     | Description                                                                      |
|-------------------------|--------------|-----------------------------|----------------------------------------------------------------------------------|
| “#” on page 231         | <i>-flag</i> | -                           | Traces the compilation without doing anything.                                   |
| “32, 64” on page 231    | <i>-qopt</i> | -                           | Selects 32- or 64-bit compiler mode.                                             |
| “aggrcopy” on page 232  | <i>-qopt</i> | See “aggrcopy” on page 232. | Enables destructive copy operations for structures and unions.                   |
| “alias” on page 233     | <i>-qopt</i> | See “alias” on page 233.    | Specifies which type-based aliasing is to be used during optimization.           |
| “align” on page 234     | <i>-qopt</i> | align=full                  | Specifies what aggregate alignment rules the compiler uses for file compilation. |
| “ansialias” on page 236 | <i>-qopt</i> | ansialias*                  | Specifies whether type-based aliasing is to be used during optimization.         |

|                                       |              |                |                                                                                                                |
|---------------------------------------|--------------|----------------|----------------------------------------------------------------------------------------------------------------|
| “arch” on page 237                    | <b>-qopt</b> | arch=com       | Specifies the architecture on which the executable program will be run.                                        |
| “assert” on page 238                  | <b>-qopt</b> | noassert       | Requests the compiler to apply aliasing assertions to your compilation unit.                                   |
| “attr” on page 238                    | <b>-qopt</b> | noattr         | Produces a compiler listing that includes an attribute listing for all identifiers.                            |
| “B” on page 239                       | <b>-flag</b> | -              | Determines substitute path names for the compiler, assembler, linkage editor, and preprocessor.                |
| “bitfields” on page 240               | <b>-flag</b> | unsigned       | Specifies if bitfields are signed.                                                                             |
| “brtl” on page 240                    | <b>-flag</b> | -              | Tells the linkage editor to accept both <b>.so</b> and <b>.a</b> library file types.                           |
| “bstatic, bdynamic” on page 241       | <b>-flag</b> | bdynamic       | Determines which types of library files are searched by the linkage editor.                                    |
| “C” on page 242                       | <b>-flag</b> | -              | Preserves comments in preprocessed output.                                                                     |
| “c” on page 242                       | <b>-flag</b> | -              | Instructs the compiler to pass source files to the compiler only.                                              |
| “cache” on page 243                   | <b>-qopt</b> | -              | Specifies the cache configuration for a specific execution machine..                                           |
| “chars” on page 244                   | <b>-qopt</b> | chars=unsigned | Instructs the compiler to treat all variables of type <b>char</b> as either <b>signed</b> or <b>unsigned</b> . |
| “check” on page 245                   | <b>-qopt</b> | nocheck        | Generates code which performs certain types of run-time checking.                                              |
| “compact” on page 246                 | <b>-qopt</b> | nocompact      | When used with optimization, reduces code size where possible, at the expense of execution speed.              |
| “cpluscmt” on page 247                | <b>-qopt</b> | nocpluscmt     | Use this option if you want C++ comments to be recognized in C source files.                                   |
| “D” on page 250                       | <b>-flag</b> | -              | Defines the identifier <i>name</i> as in a <b>#define</b> preprocessor directive.                              |
| “datalocal, dataimported” on page 251 | <b>-qopt</b> | dataimported   | Mark data as local or imported.                                                                                |

|                        |              |                          |                                                                                                                                                               |
|------------------------|--------------|--------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| “dbxextra” on page 252 | <b>-qopt</b> | nodbxextra               | Specifies that all <b>typedef</b> declarations, <b>struct</b> , <b>union</b> , and <b>enum</b> type definitions are included for debugger processing.         |
| “digraph” on page 252  | <b>-qopt</b> | nodigraph                | Allows use of digraph character sequences in your program.                                                                                                    |
| “dollar” on page 253   | <b>-qopt</b> | nodollar                 | Allows the \$ symbol to be used in the names of identifiers.                                                                                                  |
| “dpcl” on page 253     | <b>-qopt</b> | nodpcl                   | Generates symbols that tools based on the Dynamic Probe Class Library (DPCL) can use to see the structure of an executable file.                              |
| “E” on page 253        | <b>-flag</b> | -                        | Runs the source files named in the compiler invocation through the preprocessor.                                                                              |
| “enum” on page 255     | <b>-qopt</b> | enum=int                 | Specifies the amount of storage occupied by the enumerations.                                                                                                 |
| “extchk” on page 258   | <b>-qopt</b> | noextchk                 | Generates bind-time type checking information and checks for compile-time consistency.                                                                        |
| “F” on page 259        | <b>-flag</b> | -                        | Names an alternative configuration file for <b>xlc</b> .                                                                                                      |
| “f” on page 259        | <b>-flag</b> | -                        | Linkage editor ( <b>ld</b> command) option only. Passes to the linkage editor the <i>filename</i> of a file containing a list of input files to be processed. |
| “fdpr” on page 260     | <b>-qopt</b> | nofdpr                   | Collect program information for use with the AIX <b>fdpr</b> performance-tuning utility.                                                                      |
| “flag” on page 261     | <b>-qopt</b> | flag=i:i                 | Specifies the minimum severity level of diagnostic messages to be reported.                                                                                   |
| “float” on page 261    | <b>-qopt</b> | See “float” on page 261. | Specifies various floating point options to speed up or improve the accuracy of floating point operations.                                                    |
| “flttrap” on page 264  | <b>-qopt</b> | noflttrap                | Generates extra instructions to detect and trap floating point exceptions.                                                                                    |
| “fold” on page 265     | <b>-qopt</b> | fold                     | Specifies that constant floating point expressions are to be evaluated at compile time.                                                                       |



|                         |              |             |                                                                                                                                        |
|-------------------------|--------------|-------------|----------------------------------------------------------------------------------------------------------------------------------------|
| “fullpath” on page 266  | <b>-qopt</b> | nofullpath  | Specifies what path information is stored for files when you use the <b>-g</b> option.                                                 |
| “G” on page 266         | <b>-flag</b> | -           | Linkage editor ( <b>ld</b> command) option only. Used to generate a dynamic library file.                                              |
| “g” on page 267         | <b>-flag</b> | -           | Generates debugging information used by the debugger.                                                                                  |
| “genpcomp” on page 267  | <b>-qopt</b> | nogenpcomp  | Generates a precompiled version of any header file for which the original source is used.                                              |
| “genproto” on page 268  | <b>-qopt</b> | nogenproto  | Produces ANSI prototypes from K&R function definitions.                                                                                |
| “halt” on page 269      | <b>-qopt</b> | halt=s      | Instructs the compiler to stop after the compilation phase when it encounters errors of specified <i>severity</i> or greater.          |
| “heapdebug” on page 270 | <b>-qopt</b> | noheapdebug | Enables debug versions of memory management functions.                                                                                 |
| “hsflt” on page 271     | <b>-qopt</b> | nohsflt     | Speeds up calculations by removing range checking on single-precision float results and on conversions from floating point to integer. |
| “hssngl” on page 272    | <b>-qopt</b> | nohssngl    | Specifies that single-precision expressions are rounded only when the results are stored into float memory locations.                  |
| “I” on page 272         | <b>-flag</b> | -           | Specifies an additional search path if the file name in the <b>#include</b> directive is not specified using its absolute path name.   |
| “idirfirst” on page 273 | <b>-qopt</b> | noidirfirst | Specifies the search order for files included with the <b>#include</b> “ <i>file_name</i> ” directive.                                 |
| “ignerrno” on page 274  | <b>-qopt</b> | noignerrno  | Allows the compiler to perform optimizations that assume <b>errno</b> is not modified by system calls.                                 |
| “ignprag” on page 274   | <b>-qopt</b> | -           | Instructs the compiler to ignore certain pragmas.                                                                                      |
| “info” on page 275      | <b>-qopt</b> | noinfo      | Produces informational messages.                                                                                                       |

|                                   |              |                                                               |                                                                                                                                                          |
|-----------------------------------|--------------|---------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| "initauto" on page 276            | <b>-qopt</b> | noinitauto                                                    | Initializes automatic storage to the two-digit hexadecimal byte value <i>hex_value</i> .                                                                 |
| "inlglue" on page 277             | <b>-qopt</b> | 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" on page 277              | <b>-qopt</b> | See "inline" on page 277.                                     | Attempts to inline functions instead of generating calls to a function.                                                                                  |
| "ipa" on page 279                 | <b>-qopt</b> | object ( <i>compile-time</i> ),<br>noipa ( <i>link-time</i> ) | Turns on or customizes a class of optimizations known as interprocedural analysis (IPA).                                                                 |
| "isolated_call" on page 284       | <b>-qopt</b> | -                                                             | Specifies functions in the source file that have no side effects.                                                                                        |
| "L" on page 285                   | <b>-flag</b> | See "L" on page 285.                                          | Searches the specified directory for library files specified by the <b>-I</b> option.                                                                    |
| "l" on page 286                   | <b>-flag</b> | See "l" on page 286.                                          | Searches a specified library for linking.                                                                                                                |
| "langlvl" on page 286             | <b>-qopt</b> | langlvl=ansi*                                                 | Selects the C language level for compilation.                                                                                                            |
| "ldbl128, longdouble" on page 289 | <b>-qopt</b> | noldbl128                                                     | Increases the size of <b>long double</b> type from 64 bits to 128 bits.                                                                                  |
| "libansi" on page 290             | <b>-qopt</b> | nolibansi                                                     | Assumes that all functions with the name of an ANSI C library function are in fact the system functions.                                                 |
| "linedebug" on page 291           | <b>-qopt</b> | nolinedebug                                                   | Generates abbreviated line number and source file name information for the debugger.                                                                     |
| "list" on page 291                | <b>-qopt</b> | nolist                                                        | Produces a compiler listing that includes an object listing.                                                                                             |
| "listopt" on page 292             | <b>-qopt</b> | nolistopt                                                     | Produces a compiler listing that displays all options in effect.                                                                                         |
| "longlit" on page 292             | <b>-qopt</b> | nolonglit                                                     | Changes implicit type selection in 64-bit mode to use larger data types where possible.                                                                  |
| "longlong" on page 293            | <b>-qopt</b> | longlong*                                                     | Allows <b>long long</b> types in your program.                                                                                                           |
| "M" on page 294                   | <b>-flag</b> | -                                                             | Creates an output file that contains targets suitable for inclusion in a description file for the AIX <b>make</b> command.                               |

|                            |                     |             |                                                                                                                                             |
|----------------------------|---------------------|-------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| "ma" on page 295           | <i>-flag</i>        | -           | Substitutes inline code for calls to function <b>alloca</b> as if <b>#pragma alloca</b> directives are in the source code.                  |
| "macpstr" on page 295      | <b>-qopt</b>        | nomacpstr   | Converts Pascal string literals into null-terminated strings where the first byte contains the length of the string.                        |
| "maf" on page 297          | <b>-qopt</b>        | maf         | Specifies whether the floating-point multiply-add instructions are to be generated.                                                         |
| "makedep" on page 298      | <b>-qopt</b>        | -           | Creates an output file that contains targets suitable for inclusion in a description file for the AIX <b>make</b> command.                  |
| "maxerr" on page 299       | <b>-qopt</b>        | nomaxerr    | Instructs the compiler to halt compilation when a specified number of errors of specified or greater severity is reached.                   |
| "maxmem" on page 300       | <b>-qopt</b>        | maxmem=2048 | Limits the amount of memory used for local tables of specific, memory-intensive optimizations.                                              |
| "mbscs, dbscs" on page 301 | <b>-qopt</b>        | nombcs      | Use the <b>-qmbcs</b> option if your program contains multibyte characters.                                                                 |
| "noprint" on page 301      | <b>-qopt</b>        | -           | Suppresses listings.                                                                                                                        |
| "O, optimize" on page 302  | <b>-qopt, -flag</b> | nooptimize  | Optimizes code at a choice of levels during compilation.                                                                                    |
| "o" on page 305            | <i>-flag</i>        | -           | Specifies a name or directory for the output executable file(s) created either by the compiler or the linkage editor.                       |
| "once" on page 306         | <b>-qopt</b>        | noonce      | Avoids including a header file more than once even if it is specified in several of the files you are compiling.                            |
| "P" on page 307            | <i>-flag</i>        | -           | Preprocesses the C source files named in the compiler invocation and creates an output preprocessed source file for each input source file. |
| "p" on page 308            | <i>-flag</i>        | -           | Sets up the object files produced by the compiler for profiling.                                                                            |
| "pascal" on page 308       | <b>-qopt</b>        | nopascal    | Ignores the word <b>pascal</b> in type specifiers and function declarations.                                                                |

|                                                    |              |                      |                                                                                                                      |
|----------------------------------------------------|--------------|----------------------|----------------------------------------------------------------------------------------------------------------------|
| "pdf1, pdf2" on page 309                           | <b>-qopt</b> | nopdf1<br>nopdf2     | Tunes optimizations through Profile-Directed Feedback.                                                               |
| "pg" on page 311                                   | <b>-flag</b> | -                    | Sets up the object files for profiling, but provides more information than is provided by the <b>-p</b> option.      |
| "phsinfo" on page 312                              | <b>-qopt</b> | nophsinfo            | Reports the time taken in each compilation phase.                                                                    |
| "proclocal, procimported, procunknown" on page 312 | <b>-qopt</b> | proclocal*           | Mark functions as local, imported, or unknown.                                                                       |
| "proto" on page 313                                | <b>-qopt</b> | noproto              | Assumes all functions are prototyped.                                                                                |
| "Q" on page 314                                    | <b>-flag</b> | See "Q" on page 314. | Attempts to inline functions instead of generating calls to a function.                                              |
| "r" on page 316                                    | <b>-flag</b> | -                    | Produces a relocatable object.                                                                                       |
| "rndsngl" on page 316                              | <b>-qopt</b> | norndsngl            | Specifies that the result of each single-precision (float) operation is to be rounded to single precision.           |
| "ro" on page 317                                   | <b>-qopt</b> | ro*                  | Specifies the storage type for string literals.                                                                      |
| "roconst" on page 317                              | <b>-qopt</b> | roconst*             | Specifies the storage location for constant values.                                                                  |
| "rrm" on page 318                                  | <b>-qopt</b> | normm                | Prevents floating-point optimizations that are incompatible with run-time rounding to plus and minus infinity modes. |
| "S" on page 319                                    | <b>-flag</b> | -                    | Generates an assembly language file (.s) for each source file.                                                       |
| "showinc" on page 320                              | <b>-qopt</b> | noshowinc            | If used with the <b>-qsource</b> option, all the include files are included in the source listing.                   |
| "smp" on page 320                                  | <b>-qopt</b> | nosmp                | Specifies if and how parallelized object code is generated.                                                          |
| "source" on page 322                               | <b>-qopt</b> | nosource             | Produces a compiler listing and includes source code.                                                                |
| "spill" on page 323                                | <b>-qopt</b> | spill=512            | Specifies the size of the register allocation spill area.                                                            |
| "spnans" on page 323                               | <b>-qopt</b> | nospnans             | Generates extra instructions to detect signalling NaN on conversion from single precision to double precision.       |
| "srcmsg" on page 324                               | <b>-qopt</b> | nosrcmsg             | Adds the corresponding source code lines to the diagnostic messages in the <b>stderr</b> file.                       |

|                                |              |                                     |                                                                                                                                     |
|--------------------------------|--------------|-------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| “statsym” on page 324          | <b>-qopt</b> | nostatsym                           | Adds user-defined, non-external names that have a persistent storage class to the name list.                                        |
| “stdinc” on page 325           | <b>-qopt</b> | stdinc                              | Specifies which files are included with <b>#include</b> <i>&lt;file_name&gt;</i> and <b>#include</b> <i>“file_name”</i> directives. |
| “strict” on page 326           | <b>-qopt</b> | See “strict” on page 326.           | Turns off aggressive optimizations that have the potential to alter the semantics of your program.                                  |
| “strict_induction” on page 327 | <b>-qopt</b> | See “strict_induction” on page 327. | Disables loop induction variable optimizations that have the potential to alter the semantics of your program.                      |
| “syntaxonly” on page 327       | <b>-qopt</b> | -                                   | Causes the compiler to perform syntax checking without generating an object file.                                                   |
| “suppress” on page 328         | <b>-qopt</b> | nosuppress                          | Lets you specify warning or information messages to be suppressed in the compiler listing.                                          |
| “t” on page 329                | <b>-flag</b> | See “t” on page 329.                | Adds the prefix specified by the <b>-B</b> option to designated programs.                                                           |
| “tabsize” on page 329          | <b>-qopt</b> | tabsize=8                           | Changes the length of tabs as perceived by the compiler.                                                                            |
| “tbtable” on page 330          | <b>-qopt</b> | full*                               | Sets traceback table characteristics.                                                                                               |
| “threaded” on page 331         | <b>-qopt</b> | See “threaded” on page 331.         | Indicates to the compiler that the program will run in a multi-threaded environment.                                                |
| “tune” on page 331             | <b>-qopt</b> | See “tune” on page 331.             | Specifies the architecture for which the executable program is optimized.                                                           |
| “U” on page 332                | <b>-flag</b> | -                                   | Undefines a specified identifier defined by the compiler or by the <b>-D</b> option.                                                |
| “unroll” on page 333           | <b>-qopt</b> | unroll=4*                           | Unrolls inner loops in the program by a specified factor.                                                                           |
| “upconv” on page 334           | <b>-qopt</b> | noupconv*                           | Preserves the <b>unsigned</b> specification when performing integral promotions.                                                    |

|                       |              |           |                                                                                                            |
|-----------------------|--------------|-----------|------------------------------------------------------------------------------------------------------------|
| "usecomp" on page 335 | <b>-qopt</b> | nousecomp | Use precompiled header files for any files that have not changed since the precompiled header was created. |
| "v" on page 336       | <b>-flag</b> | -         | Instructs the compiler to report information on the progress of the compilation.                           |
| "W" on page 336       | <b>-flag</b> | -         | Passes the listed words to a designated compiler program.                                                  |
| "w" on page 337       | <b>-flag</b> | -         | Requests that warning messages be suppressed.                                                              |
| "warn64" on page 338  | <b>-qopt</b> | nowarn64  | Enables warning of possible long to integer data truncations.                                              |
| "xcall" on page 338   | <b>-qopt</b> | noxcall   | Generates code to static routines within a compilation unit as if they were external calls.                |
| "xref" on page 339    | <b>-qopt</b> | noxref    | Produces a compiler listing that includes a cross-reference listing of all identifiers.                    |
| "y" on page 339       | <b>-flag</b> | -         | Specifies the compile-time rounding mode of constant floating-point expressions.                           |

### Related Reading

- "Invoking the Compiler" on page 8
- "Specifying Compiler Options on the Command Line" on page 10
- "Specifying Compiler Options in Your Program Source Files" on page 12
- "Specifying Compiler Options in a Configuration File" on page 13
- "Options that Specify Compiler Characteristics"
- "Options that Specify Debugging Features" on page 227
- "Options that Specify Preprocessor Options" on page 228
- "Options that Specify Compiler Output" on page 228
- "Options that Specify the Compiler Object Code Produced" on page 229
- "Options that Specify Linkage Options" on page 230
- "Resolving Conflicting Compiler Options" on page 217

---

## Lists of Compiler Options by Functional Groupings

### Options that Specify Compiler Characteristics

| To:                                              | See:                  |
|--------------------------------------------------|-----------------------|
| Specify the language level                       | "langlvl" on page 286 |
| Specify a different configuration file or stanza | "F" on page 259       |
| Specify path names to other program names        | "B" on page 239       |
| Specify program options                          | "W" on page 336       |
| Specify a search path                            | "I" on page 272       |

|                                                                                  |                                             |
|----------------------------------------------------------------------------------|---------------------------------------------|
| Specify if <b>char</b> variables are treated as <b>signed</b> or <b>unsigned</b> | "chars" on page 244                         |
| Specify the use of multibyte characters                                          | "mbcs, dbcs" on page 301                    |
| Change the length of tabs in your source file                                    | "tabsize" on page 329                       |
| Produce ANSI prototypes from K&R function definitions                            | "genproto" on page 268                      |
| Specify aliasing assertions                                                      | "alias" on page 233<br>"assert" on page 238 |

### Related Reading

- "Invoking the Compiler" on page 8
- "Specifying Compiler Options on the Command Line" on page 10
- "Specifying Compiler Options in Your Program Source Files" on page 12
- "Specifying Compiler Options in a Configuration File" on page 13
- "Compiler Options and Their Defaults" on page 218
- "Options that Specify Debugging Features"
- "Options that Specify Preprocessor Options" on page 228
- "Options that Specify Compiler Output" on page 228
- "Options that Specify the Compiler Object Code Produced" on page 229
- "Options that Specify Linkage Options" on page 230

## Options that Specify Debugging Features

| To:                                                                                        | See:                                |
|--------------------------------------------------------------------------------------------|-------------------------------------|
| Produce only line number and source file name information for <b>dbx</b>                   | "linedebug" on page 291             |
| Produce debug information for <b>dbx</b>                                                   | "g" on page 267                     |
| Generates symbols for use by tools based on the Dynamic Probe Class Library (DPCL)         | "dpcl" on page 253                  |
| Enable debug versions of memory management functions                                       | "heapdebug" on page 270             |
| Specify full path information when you use " <b>g</b> " on <b>page 267</b> with <b>dbx</b> | "fullpath" on page 266              |
| Generate and set the characteristics of the traceback table                                | "tbtable" on page 330               |
| Set up object files for profiling                                                          | "p" on page 308<br>"pg" on page 311 |
| Trap division of an integer by zero                                                        | "check" on page 245                 |
| Ignore " <b>isolated_call</b> " on <b>page 284</b> aliasing pragmas                        | "ignprag" on page 274               |

### Related Reading

- "Debugging Memory Heaps" on page 183
- "Memory Management Functions" on page 179
- "Managing Memory with Multiple Heaps" on page 182
- "Debugging Programs with Heap Memory" on page 195
- "Invoking the Compiler" on page 8
- "Specifying Compiler Options on the Command Line" on page 10
- "Specifying Compiler Options in Your Program Source Files" on page 12
- "Specifying Compiler Options in a Configuration File" on page 13
- "Compiler Options and Their Defaults" on page 218
- "Options that Specify Compiler Characteristics" on page 226
- "Options that Specify Preprocessor Options" on page 228
- "Options that Specify Compiler Output" on page 228
- "Options that Specify the Compiler Object Code Produced" on page 229
- "Options that Specify Linkage Options" on page 230

## Options that Specify Preprocessor Options

| To:                                                        | See:                                     |
|------------------------------------------------------------|------------------------------------------|
| Define a name in a <b>#define</b> directive                | "D" on page 250                          |
| Undefine a name as in a <b>#undef</b> directive            | "U" on page 332                          |
| Create an output file for use with the <b>make</b> command | "M" on page 294<br>"makedep" on page 298 |

### Related Reading

- "Invoking the Compiler" on page 8
- "Specifying Compiler Options on the Command Line" on page 10
- "Specifying Compiler Options in Your Program Source Files" on page 12
- "Specifying Compiler Options in a Configuration File" on page 13
- "Compiler Options and Their Defaults" on page 218
- "Options that Specify Compiler Characteristics" on page 226
- "Options that Specify Debugging Features" on page 227
- "Options that Specify Compiler Output"
- "Options that Specify the Compiler Object Code Produced" on page 229
- "Options that Specify Linkage Options" on page 230

## Options that Specify Compiler Output

| To:                                                                                                   | See:                                                                                                                                                                                     |
|-------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Perform syntax checking but do not generate an object file                                            | "syntaxonly" on page 327                                                                                                                                                                 |
| Compile but not link                                                                                  | "c" on page 242<br>"C" on page 242<br>"E" on page 253<br>"P" on page 307                                                                                                                 |
| Create a dynamic library object file ( <b>ld</b> command only)                                        | "G" on page 266                                                                                                                                                                          |
| Suppress output listings                                                                              | "C" on page 242                                                                                                                                                                          |
| Produce compiler listings                                                                             | "noprnt" on page 301<br>"source" on page 322<br>"showinc" on page 320<br>"srcmsg" on page 324<br>"xref" on page 339<br>"attr" on page 238<br>"list" on page 291<br>"listopt" on page 292 |
| Specify severity level of diagnostic messages                                                         | "flag" on page 261                                                                                                                                                                       |
| Suppress messages                                                                                     | "suppress" on page 328<br>"w" on page 337                                                                                                                                                |
| Halt the compiler output if errors of specified <i>severity</i> or greater are encountered            | "halt" on page 269                                                                                                                                                                       |
| Halt the compiler output if <i>num</i> errors of specified or greater <i>severity</i> are encountered | "maxerr" on page 299                                                                                                                                                                     |
| Produce information messages                                                                          | "info" on page 275                                                                                                                                                                       |
| Report the time taken for compilation                                                                 | "phsinfo" on page 312                                                                                                                                                                    |
| Report status information as the compilation proceeds                                                 | "v" on page 336                                                                                                                                                                          |
| Trace the compilation                                                                                 | "#" on page 231                                                                                                                                                                          |



## Related Reading

- “Invoking the Compiler” on page 8
- “Specifying Compiler Options on the Command Line” on page 10
- “Specifying Compiler Options in Your Program Source Files” on page 12
- “Specifying Compiler Options in a Configuration File” on page 13
- “Compiler Options and Their Defaults” on page 218
- “Options that Specify Compiler Characteristics” on page 226
- “Options that Specify Debugger Features” on page 227
- “Options that Specify Preprocessor Options” on page 228
- “Options that Specify the Compiler Object Code Produced”
- “Options that Specify Linkage Options” on page 230

## Options that Specify the Compiler Object Code Produced

| To:                                                                               | See:                                                                                                                                                |
|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| Specify the architecture on which the executable program will be run              | “32, 64” on page 231<br>“arch” on page 237<br>“tune” on page 331                                                                                    |
| Use the linkage editor to create a dynamic library file ( <b>ld</b> command only) | “G” on page 266                                                                                                                                     |
| Specify the register allocation spill area                                        | “spill” on page 323                                                                                                                                 |
| Specify if and how parallelized object code is generated.                         | “smp” on page 320                                                                                                                                   |
| Choose code optimization options                                                  | “cache” on page 243<br>“aggrcopy” on page 232<br>“O, optimize” on page 302<br>“pdf1, pdf2” on page 309<br>“ipa” on page 279<br>“unroll” on page 333 |
| Generate information used by the <b>fdpr</b> performance-tuning utility           | “fdpr” on page 260                                                                                                                                  |
| Set inlining options                                                              | “Q” on page 314<br>“ipa” on page 279<br>“inline” on page 277                                                                                        |
| Choose alignment rules for aggregates                                             | “align” on page 234                                                                                                                                 |
| Choose storage type for constant values                                           | “roconst” on page 317                                                                                                                               |
| Choose storage types for string literals                                          | “ro” on page 317                                                                                                                                    |
| Set the size of a <b>long double</b> (64 or 128 bits)                             | “ldbl128, longdouble” on page 289                                                                                                                   |
| Ignore <b>long long int</b> types                                                 | “longlong” on page 293                                                                                                                              |
| Set the rounding mode of floating-point expressions                               | “y” on page 339                                                                                                                                     |
| Reduce code size                                                                  | “compact” on page 246                                                                                                                               |
| Set floating point options                                                        | “float” on page 261                                                                                                                                 |
| Set rounding of single-precision expressions                                      | “float” on page 261 ( <b>-qfloat=hssngl</b> )                                                                                                       |
| Include extra instructions to detect NaN                                          | “float” on page 261 ( <b>-qfloat=nans</b> )                                                                                                         |
| Remove range checking                                                             | “float” on page 261 ( <b>-qfloat=hsflt</b> )                                                                                                        |
| Set rounding of single-precision ( <b>float</b> ) operations                      | “float” on page 261 ( <b>-qfloat=rndsngl</b> )                                                                                                      |
| Detect and trap floating point exceptions                                         | “flttrap” on page 264                                                                                                                               |
| Generate floating point multiply-add instructions                                 | “float” on page 261 ( <b>-qfloat=maf</b> )                                                                                                          |
| Prevent incompatible optimizations                                                | “float” on page 261 ( <b>-qfloat=rrm</b> )                                                                                                          |
| Evaluate floating point expressions at compile time                               | “float” on page 261 ( <b>-qfloat=fold</b> )                                                                                                         |

|                                                                                |                                                    |
|--------------------------------------------------------------------------------|----------------------------------------------------|
| Generate bind-time type checking                                               | "extchk" on page 258                               |
| Choose type-based aliasing during optimization                                 | "alias" on page 233<br>"ansialias" on page 236     |
| Initialize automatic storage                                                   | "initauto" on page 276                             |
| Limit the amount of memory                                                     | "maxmem" on page 300                               |
| Mark data as local or imported                                                 | "datalocal, dataimported" on page 251              |
| Mark functions as local, imported, or unknown                                  | "proclocal, procimported, procunknown" on page 312 |
| Substitute inline code for calls to <b>alloca</b>                              | "ma" on page 295                                   |
| Perform optimizations that assume <b>errno</b> is not modified by system calls | "ignerrno" on page 274                             |

### Related Reading

- "Invoking the Compiler" on page 8
- "Specifying Compiler Options on the Command Line" on page 10
- "Specifying Compiler Options in Your Program Source Files" on page 12
- "Specifying Compiler Options in a Configuration File" on page 13
- "Compiler Options and Their Defaults" on page 218
- "Options that Specify Compiler Characteristics" on page 226
- "Options that Specify Debugging Features" on page 227
- "Options that Specify Preprocessor Options" on page 228
- "Options that Specify Compiler Output" on page 228
- "Options that Specify Linkage Options"

## Options that Specify Linkage Options

| To:                                                                | See:                                                  |
|--------------------------------------------------------------------|-------------------------------------------------------|
| Name the output file or directory                                  | "o" on page 305                                       |
| Search specified libraries                                         | "l" on page 286                                       |
| Search a path for libraries                                        | "L" on page 285                                       |
| Produce an output file even if not all symbols are resolved        | "r" on page 316                                       |
| Specify which types of library file are used by the linkage editor | "brtl" on page 240<br>"bstatic, bdynamic" on page 241 |
| Generate fast external linkage                                     | "inlglue" on page 277                                 |

### Related Reading

- "Invoking the Compiler" on page 8
- "Specifying Compiler Options on the Command Line" on page 10
- "Specifying Compiler Options in Your Program Source Files" on page 12
- "Specifying Compiler Options in a Configuration File" on page 13
- "Compiler Options and Their Defaults" on page 218
- "Options that Specify Compiler Characteristics" on page 226
- "Options that Specify Debugging Features" on page 227
- "Options that Specify Preprocessor Options" on page 228
- "Options that Specify Compiler Output" on page 228
- "Options that Specify the Compiler Object Code Produced" on page 229

---

## Compiler Options Reference

### #

| Option Type  | Default Value | #pragma options |
|--------------|---------------|-----------------|
| <i>-flag</i> | -             | -               |

#### Syntax

-#

#### Purpose

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

#### Notes:

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

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

#### Example

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

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

#### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Output” on page 228

“**v**” on page 336

## 32, 64

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <i>-qoption</i> | -             | -               |

#### Syntax

-q32 | -q64

#### Purpose

Selects either 32- or 64-bit compiler mode.

#### 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 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**” on page 16 for valid combinations of the **-q32**, **-q64**, **-qarch**, and **-qtune** compiler options.

If specified alone without accompanying **-qarch** and **-qtune** compiler options, the C for AIX compiler treats:

- **-q32** as **-qarch=com -q32**
- **-q64** as **-qarch=com -q64**

### 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, 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 Reading

“Specifying Compiler Options for Architecture-Specific, 32- or 64-bit Compilation” on page 14

“Acceptable Compiler Mode and Processor Architecture Combinations” on page 16

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

“arch” on page 237

“tune” on page 331

## aggrcopy

| Option Type     | Default Value     | #pragma options |
|-----------------|-------------------|-----------------|
| <b>-qoption</b> | <i>See notes.</i> | -               |

### Syntax

```
-qaggrcopy=overlap | -qaggrcopy=nooverlap
```

### Purpose

Enables destructive copy operations for structures and unions.

### 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.

### Default Setting

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

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

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

## Example

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

## Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

## alias

| Option Type           | Default Value                       | #pragma options                               |
|-----------------------|-------------------------------------|-----------------------------------------------|
| <code>-qoption</code> | ansi:typeptr:noallptrs:noaddrtaken* | ALIAS= <i>suboption</i> [: <i>suboption</i> ] |

## Syntax

```
-qalias=suboption[:suboption][...]  
ALIAS=suboption[:suboption]
```

## 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, unless you specify otherwise.

## Notes

If used, **#pragma ALIAS=*suboption*** must appear before the first program statement.

The compiler will apply aliasing assertions according to the following *suboptions*:

|               |                                                                                                                                                                                                                                                                                                                                                                                  |
|---------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [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 <b>-qalias=typeptr</b> ). 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 ( <b>ansi</b> ) is the default for the <b>xlc</b> and <b>c89 compilers</b> . This option has no effect unless you also specify the <b>-O</b> 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.

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 Reading

- “Compiler Options and Their Defaults” on page 218
- “Options that Specify the Compiler Object Code Produced” on page 229
- “Options that Specify Compiler Characteristics” on page 226
- “O, optimize” on page 302

## align

| Option Type     | Default Value | #pragma options         |
|-----------------|---------------|-------------------------|
| <b>-qoption</b> | align=full    | ALIGN= <i>suboption</i> |

### Syntax

```
-qalign=suboption  
ALIGN=suboption
```

### 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.

### Notes

The **-qalign** suboptions are:

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

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.

### Example 1 - Imbedded #pragmas

Using the compiler invocation:

```
xlC -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:

```
xlc 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 */
```

### Example 3 - Sample bit\_packed Fields

Assuming the following structure is declared:

```
#pragma options align=bit_packed
struct {
    int a : 8;
    int c : 12;
    int d : 4;
    int e : 3;
    int : 0;
    int f : 1;
    char g;
} A;
#pragma options align=reset;
```

The structure takes on the following characteristics:

sizeof(A) = 7

and the layout of A is:

| Member Name | Offset |      |
|-------------|--------|------|
|             | Bytes  | Bits |
| a           | 0      | 0    |
| b           | 1      | 0    |
| c           | 2      | 2    |

|   |   |   |
|---|---|---|
| d | 3 | 6 |
| e | 4 | 2 |
| f | 5 | 0 |
| g | 6 | 0 |

Note that there is no padding between bitfield members c and d.

### Related Reading

- “`__align` Specifier” on page 442
- “Compiler Options and Their Defaults” on page 218
- “Options that Specify the Compiler Object Code Produced” on page 229
- “Appendix H. RISC System/6000 Alignment Rules” on page 437
- “MacIntosh and Twobyte Alignment Rules” on page 440
- “Packed Alignment Rules” on page 438
- “Alignment Rules for Nested Aggregates” on page 438

## ansialias

| Option Type           | Default Value | #pragma options |
|-----------------------|---------------|-----------------|
| <code>-qoption</code> | ansialias*    | ANSIALIAS       |

### Syntax

```
-qansialias | -qnoansialias
ANSIALIAS | NOANSIALIAS
```

### 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.

### Notes

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

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

\* The default with `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`.

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 Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

“O, optimize” on page 302

“alias” on page 233

## arch

| Option Type           | Default Value | #pragma options |
|-----------------------|---------------|-----------------|
| <code>-qoption</code> | arch=com      | -               |

### Syntax

`-qarch=suboption`

### Purpose

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

### Notes

If you want maximum performance on a specific architecture and will not be using the program on other architectures, use the appropriate processor architecture option. You can specify the architecture using the following basic suboptions:

|        |                                                                                                                                                                                                                                                                                                                         |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| auto   | Automatically detects the specific architecture of the compiling machine. Use this suboption only if the execution environment is the same as the compilation environment.                                                                                                                                              |
| com    | Produces object code that contains instructions that will run on all the POWER, POWER2*, and PowerPC* hardware platforms (that is, the instructions generated are <i>common</i> to all platforms. Using <b>-qarch=com</b> is referred to as compiling in <i>common mode</i> ) Defines the <code>_ARCH_COM</code> macro. |
|        | Use this option if you want your program to be portable.                                                                                                                                                                                                                                                                |
| pwr    | Produces object code that contains instructions that will run on any of the POWER and POWER2 hardware platforms. Defines the <code>_ARCH_PWR</code> macro.                                                                                                                                                              |
| pwr2   | Produces object code that contains instructions that will run on the POWER2 hardware platforms. Defines the <code>_ARCH_PWR</code> and <code>_ARCH_PWR2</code> macros.                                                                                                                                                  |
| pwrx   | Produces object code that contains instructions that will run on the POWER2 hardware platforms (same as <b>-qarch=pwr2</b> ). Defines the <code>_ARCH_PWR</code> and <code>_ARCH_PWR2</code> macros.                                                                                                                    |
| ppc    | Produces object code that contains 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. Defines the <code>_ARCH_PPC</code> macro.                                         |
| ppcgr  | Produces object code that contains optional graphics instructions for PowerPC processors. Defines the <code>_ARCH_PPC</code> and <code>_ARCH_PPCGR</code> macros.                                                                                                                                                       |
| noauto | Valid only when the <b>-O4</b> compiler option is in effect, this option disables automatic setting of the <b>-qarch</b> and <b>qtune</b> compiler options.                                                                                                                                                             |

Additional **-qarch** suboptions for specific processors can be found in “Acceptable Compiler Mode and Processor Architecture Combinations” on page 16.

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.

### Default

The default setting of the **-qarch** option depends on the setting of the **-qtune** 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 “Acceptable Compiler Mode and Processor Architecture Combinations” on page 16.

### 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 Reading

“Specifying Compiler Options for Architecture-Specific, 32- or 64-bit Compilation” on page 14

“Acceptable Compiler Mode and Processor Architecture Combinations” on page 16

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

“O, optimize” on page 302

“tune” on page 331

## assert

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | noassert      | -               |

### Syntax

```
-qassert=suboption
```

### 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.

### Notes

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

The compiler will apply aliasing assertions when you specify the following *suboptions*:

|                           |                                                                                                                                                                                                                                        |
|---------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>-qASsert=TYPEptr</b>   | 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.                                                                     |
| <b>-qASsert=ALLPtrs</b>   | Pointers are never aliased (this implies <b>-qassert=typeptr</b> ). Therefore, in the compilation unit, no two pointers will point to the same storage location.                                                                       |
| <b>-qASsert=ADDRtaken</b> | 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. |

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Characteristics” on page 226

“alias” on page 233

## attr

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | noattr        | ATTR            |

## Syntax

`-qattr` | `-qattr=full` | `-qnoattr`  
`ATTR` | `ATTR=FULL` | `NOATTR`

## Purpose

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

## Notes

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

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

The `-qnoprint` option overrides this option.

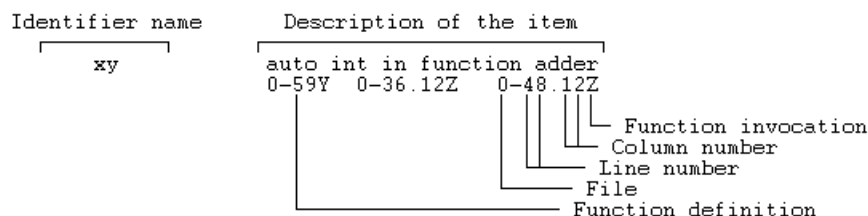
If `-qattr` is specified after `-qattr=full`, it has no effect. The full listing is produced.

## Example

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

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

A typical cross-reference listing has the form:



## Related Reading

“Compiler Options and Their Defaults” on page 218  
“Options that Specify Compiler Output” on page 228  
“noprint” on page 301  
“xref” on page 339

## B

| Option Type        | Default Value | #pragma options |
|--------------------|---------------|-----------------|
| <code>-flag</code> | -             | -               |

## Syntax

`-B` | `-Bprefix` | `-B -tprograms` | `-Bprefix -tprograms`

## Purpose

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

## Notes

The optional *prefix* defines part of a path name to the new programs. It must end in `/`.

To form the complete path name for each program, the C for AIX compiler 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 the C for AIX compiler executables and have the option of specifying which one you want to use.

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

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

The **-Bprefix -tprograms** options override the **-Fconfig\_file** option.

### Example

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

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

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

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

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Characteristics” on page 226

“F” on page 259

“t” on page 329

## bitfields

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | unsigned      | -               |

### Syntax

```
-qbitfields=suboption
```

### Purpose

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

### Notes

The **-qbitfields** suboptions are:

|          |                         |
|----------|-------------------------|
| signed   | Bitfields are signed.   |
| unsigned | Bitfields are unsigned. |

### Related Reading

“Compiler Options and Their Defaults” on page 218

## brtl

| Option Type  | Default Value | #pragma options |
|--------------|---------------|-----------------|
| <b>-flag</b> | -             | -               |

### Syntax

```
-brtl
```

### Purpose

Tells the linkage editor to perform library searches of both **.a** and **.so** library files.

## Notes

This option affects all library searches.

For a library name and path specified by the **-I** and **-L** options, the linkage editor searches, if they exist, the **.so** library first and then the **.a** library.

## Example

To compile `myprogram.c` searching both shared and static versions of the C for AIX compiler libraries, enter:

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

## Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Linkage Options” on page 230

“**bstatic**, **bdynamic**”

“**I**” on page 286

“**L**” on page 285

## **bstatic**, **bdynamic**

| Option Type  | Default Value         | #pragma options |
|--------------|-----------------------|-----------------|
| <i>-flag</i> | <code>bdynamic</code> | -               |

## Syntax

```
-bstatic | -bdynamic
```

## Purpose

Controls how libraries are processed by specifying which forms of library names the linkage editor looks for.

## Notes

The linkage editor searches library names and paths specified by the **-I** and **-L** options according to the following criteria:

|                       |                                                                                                                                                                                                                                                                                                                                                     |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>bdynamic</code> | For settings of the <b>-Ikey</b> option appearing after the <b>-bdynamic</b> option, both <b>libkey.so</b> and <b>libkey.a</b> library files are searched for by the linkage editor. This option remains in effect until overridden by the appearance of the <b>-bstatic</b> option, which in turn affects <b>-Ikey</b> options appearing after it. |
| <code>bstatic</code>  | For settings of the <b>-Ikey</b> option appearing after the <b>-bstatic</b> option, only <b>libkey.a</b> library files are searched for by the linkage editor. This option remains in effect until overridden by the appearance of the <b>-bdynamic</b> option, which in turn affects <b>-Ikey</b> options appearing after it.                      |

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.

## Example

To compile `myprogram.c` using a static version of the **libtask.a** Task Library and a dynamic version version of the **libcomplex.a** Complex Mathematics Library, enter:

```
xlc myprogram.c -bstatic -ltask -bdynamic -lcomplex
```

## Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Linkage Options” on page 230

“**I**” on page 286

“**L**” on page 285

## C

| Option Type  | Default Value | #pragma options |
|--------------|---------------|-----------------|
| <i>-flag</i> | -             | -               |

### Syntax

-C

### Purpose

Preserves comments in preprocessed output.

### 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 Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Output” on page 228

“C”

“E” on page 253

“P” on page 307

## C

| Option Type  | Default Value | #pragma options |
|--------------|---------------|-----------------|
| <i>-flag</i> | -             | -               |

### Syntax

-c

### Purpose

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

### Notes

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

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

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

### Example

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

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

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

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

## Related Reading

“Compiler Options and Their Defaults” on page 218  
“Options that Specify Compiler Output” on page 228  
“E” on page 253  
“o” on page 305  
“P” on page 307  
“syntaxonly” on page 327

## cache

| Option Type           | Default Value | #pragma options |
|-----------------------|---------------|-----------------|
| <code>-qoption</code> | -             | -               |

### Syntax

```
-qcache=  
{  
  assoc=number |  
  auto |  
  cost=cycles |  
  level=level |  
  line=bytes |  
  size=Kbytes |  
  type=cache_type  
}[: ...]
```

### Purpose

Use this option to describe the cache configuration for a specific target execution machine, if different from the compiling machine. The compiler uses this configuration information to optimize program performance, particularly loop operations that can be structured or blocked, to maximize effective use of the data cache on the target execution machine.

### Notes

The `-qcache` option has an effect only if you also specify the `-qipa`, `-O4`, `-O5`, or `-qsmp` options.

| Suboption                        | Description                                                                                                                                                                                                                                                                                                                                                                                                               |
|----------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>assoc=<i>number</i></code> | Specifies the set associativity of the cache, where <i>number</i> can be:<br><br><b>0</b> Direct-mapped cache<br><br><b>1</b> Fully-associative cache<br><br><b><i>n</i> &gt; 1</b> <i>n</i> -way set-associative cache                                                                                                                                                                                                   |
| <code>auto</code>                | Specifies the cache configuration to match that of the compiling machine.                                                                                                                                                                                                                                                                                                                                                 |
| <code>cost=<i>cycles</i></code>  | Specifies in instruction <i>cycles</i> the estimated performance penalty that results from a cache miss. The compiler uses this value when deciding whether or not to perform an optimization that might result in extra cache misses.                                                                                                                                                                                    |
| <code>level=<i>level</i></code>  | Specifies the level of cache affected, where <i>level</i> can be:<br><br><b>1</b> Level-1 cache<br><br><b>2</b> Level-2 cache, or the translation look-aside buffer in a machine that has no level-2 cache.<br><br><b>3</b> Translation look-aside buffer in a machine that has a level-2 cache.<br><br>If a machine has more than one level of cache, use a separate <code>-qcache</code> option to describe each cache. |
| <code>line=<i>bytes</i></code>   | Specifies the line size of the cache in <i>bytes</i> .                                                                                                                                                                                                                                                                                                                                                                    |
| <code>size=<i>Kbytes</i></code>  | Specifies the total size of the cache in <i>Kbytes</i> .                                                                                                                                                                                                                                                                                                                                                                  |

| Suboption                    | Description                                                                                                                                                                                                                                                                   |
|------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>type=cache_type</code> | Specifies the type of cache to which the above settings apply, where <i>cache_type</i> can be: <ul style="list-style-type: none"> <li><b>C or c</b> Combined data and instruction cache</li> <li><b>D or d</b> Data cache</li> <li><b>I or i</b> Instruction cache</li> </ul> |

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.

### Examples

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

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

2. To tune performance for a system with two levels of data cache, specify the **-qcache** option once for each level of cache:

```
xlc -O4 -qcache=type=D:level=1:size=256:line=256:assoc=4 \
-qcache=type=D:level=2:size=512:line=256:assoc=2 file.c
```

3. To tune performance for a system with two types of cache, again, specify the **-qcache** option once for each type of cache:

```
xlc -O5 -qipa -qcache=type=D:level=1:size=256:line=256:assoc=4 \
-qcache=type=I:level=1:size=512:line=256:assoc=2 file.c
```

### Related Reading

- “Compiler Options and Their Defaults” on page 218
- “Options that Specify the Compiler Object Code Produced” on page 229
- “O, optimize” on page 302
- “arch” on page 237
- “ipa” on page 279
- “smp” on page 320
- “tune” on page 331

## chars

| Option Type     | Default Value  | #pragma options         |
|-----------------|----------------|-------------------------|
| <b>-qoption</b> | chars=unsigned | CHARS= <i>sign_type</i> |

### Syntax



```
-qchars=signed | -qchars=unsigned
CHARS=signed | CHARS=unsigned
```

### Purpose

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

### 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**.

The `_CHAR_SIGNED` or `_CHAR_UNSIGNED` macros are defined according to the setting of the **-qchars** option or corresponding preprocessor directives.

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.

### Example

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

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

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Characteristics” on page 226

## check

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | nocheck       | CHECK           |

### Syntax

```
-qcheck | -qcheck=suboptions | -qnocheck
CHECK | CHECK=suboptions | NOCHECK
```

### Purpose

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

### Notes

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

**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 **-qchecks** suboptions 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.                                                                                                                                                                                     |

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.

### Example

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 */
}
```

For **-qcheck=divzero**:

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

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Debugging Features” on page 227

## compact

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | nocompact     | COMPact         |

### Syntax

```
-qcompact | -qnocompact
COMPACT | NOCOMPACT
```

### Purpose

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

### Notes

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

## Example

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

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

## Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

“#pragma option\_override Preprocessor Directive” on page 374

## cplusplus

| Option Type | Default Value | #pragma options |
|-------------|---------------|-----------------|
| -qoption    | nocplusplus   | CPLUSCMT        |

## Syntax

```
-qcplusplus | -qnocplusplus  
CPLUSCMT | NOCPLUSCMT
```

## Purpose

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

## Notes

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

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 ANSI C. The result of the following valid ANSI C 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 (**??/**).

## 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
```

### 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;
```

### 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;
```

### 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;
```

A comment may contain a sequence of valid multibyte characters.

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.

Macro replacement is not performed within comments.

### Compile Example

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

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

### Related Reading

- “Compiler Options and Their Defaults” on page 218
- “Options that Specify Preprocessor Options” on page 228
- “C” on page 242
- “E” on page 253
- “P” on page 307

## D

| Option Type  | Default Value | #pragma options |
|--------------|---------------|-----------------|
| <i>-flag</i> | -             | -               |

### Syntax

```
-Dname=definition | -Dname= | -Dname
```

### Purpose

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

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

`-Dname=` is equivalent to `#define name`.

`-Dname` 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:

- `_POWER`
- `_AIX`
- `_AIX32`
- `_IBMR2`
- `_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 **-Uname** option has a higher precedence than the **-Dname** option.

**Example** 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 Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Preprocessor Options” on page 228

“U” on page 332

## datalocal, dataimported

| Option Type     | Default Value | #pragma options         |
|-----------------|---------------|-------------------------|
| <b>-qoption</b> | dataimported  | DATALOCaI, DATAIMPoRted |

### Syntax

```
-qdatalocal | -qdatalocal=names
-qdataimported | -qdataimported=names
DATALOCAL | DATALOCAL=names
DATAIMPORTED | DATAIMPORTED=names
```

### Purpose

Mark data as local or imported.

### Notes

Local variables are statically bound with the functions that use them. **-qdatalocal** changes the default to assume that all variables are local. **-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.

Imported variables are dynamically bound with a shared portion of a library. **-qdataimported** changes the default to assume that all variables are imported. **-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 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 Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

## dbxextra

| Option Type           | Default Value | #pragma options |
|-----------------------|---------------|-----------------|
| <code>-qoption</code> | nodbxextra    | -               |

### Syntax

`-qdbxextra` | `-qnodbxextra`

### Purpose

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

### Notes

Use this option with the **-g** option to produce additional debugging information.

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 debugger processing, enter:

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

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Debugging Features” on page 227

“g” on page 267

## digraph

| Option Type           | Default Value | #pragma options |
|-----------------------|---------------|-----------------|
| <code>-qoption</code> | nodigraph     | -               |

### Syntax

`-qdigraph` | `-qnodigraph`

### Purpose

Lets you use digraph character sequences to represent characters not found on some keyboards. Digraphs are enabled by default.

### Example

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

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



### Related Reading

“Compiler Options and Their Defaults” on page 218  
“C Programming Character Set” on page 160

## dollar

| Option Type           | Default Value | #pragma options |
|-----------------------|---------------|-----------------|
| <code>-qoption</code> | nodollar      | -               |

### Syntax

`-qdollar` | `-qnodollar`

### Purpose

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

### Example

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

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

### Related Reading

“Compiler Options and Their Defaults” on page 218

## dpcl

| Option Type           | Default Value | #pragma options |
|-----------------------|---------------|-----------------|
| <code>-qoption</code> | nodpcl        | -               |

### Syntax

`-qdpcl` | `-qnodpcl`

### Purpose

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

### 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 Reading

“Compiler Options and Their Defaults” on page 218  
“Options that Specify the Compiler Object Code Produced” on page 229

## E

| Option Type        | Default Value | #pragma options |
|--------------------|---------------|-----------------|
| <code>-type</code> | -             | -               |

### Syntax

-E

## Purpose

Runs the source files named in the compiler invocation through the preprocessor. The **-E** option calls the preprocessor directly as `/usr/vac/exe/xlCcpp`.

## Notes

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

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

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

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

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

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

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

Comments are replaced in the preprocessed output by a single space character. New lines and **#line** directives are issued for comments that span multiple source lines, and when **-C** is not specified. Comments within a macro function argument are deleted.

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

## Example

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

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

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

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

the output will be:

```
#line 2 "myprogram.c"
int a;
#line 5
int b;
int c;
c =
(a + b);
```

### Related Reading

“Compiler Options and Their Defaults” on page 218  
 “Options that Specify Compiler Output” on page 228  
 “#line Preprocessor Directive” on page 357  
 “C” on page 242  
 “M” on page 294  
 “o” on page 305  
 “P” on page 307  
 “syntaxonly” on page 327

## enum

| Option Type           | Default Value | #pragma options        |
|-----------------------|---------------|------------------------|
| <code>-qoption</code> | enum=int      | ENUM= <i>suboption</i> |

### Syntax

```
-qenum=small | -qenum=int | -qenum=intlong | -qenum=1 | -qenum=2 | -qenum=4 | -qenum=8
ENUM=SMALL | ENUM=INT | ENUM=INTLONG | ENUM=1 | ENUM=2 | ENUM=4 | ENUM=8 | ENUM=RESET
```

### Purpose

Specifies the amount of storage occupied by enumerations.

### Notes

Valid *suboptions* are:

|                             |                                                                                                                                                                                                                                                                                  |
|-----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-qenum=small</code>   | Specifies that enumerations occupy a minimum amount of storage: either 1, 2, or 4 bytes of storage, depending on the range of the <b>enum</b> constants.                                                                                                                         |
| <code>-qenum=int</code>     | Specifies that enumerations occupy 4 bytes of storage and are represented by <b>int</b> .                                                                                                                                                                                        |
| <code>-qenum=intlong</code> | Specifies that enumerations occupy 8 bytes of storage and are represented by <b>long</b> , if <b>-q64</b> is specified and the range of the <b>enum</b> constants exceed the limit for <b>int</b> .                                                                              |
| <code>-qenum=1</code>       | Specifies that enumerations occupy 1 byte of storage.                                                                                                                                                                                                                            |
| <code>-qenum=2</code>       | Specifies that enumerations occupy 2 bytes of storage.                                                                                                                                                                                                                           |
| <code>-qenum=4</code>       | Specifies that enumerations occupy 4 bytes of storage.                                                                                                                                                                                                                           |
| <code>-qenum=8</code>       | Valid only in 64-bit compiler mode. Specifies that enumerations occupy 8 bytes of storage.                                                                                                                                                                                       |
| <code>RESET</code>          | Valid in <b>#pragma enum</b> statement only. Resets the <b>enum</b> mapping rule to the rule that was in effect before the current mapping rule. If no previous enum mapping rule was specified in the file, the rule specified when the compiler was initially invoked is used. |

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 **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=1|2|4** 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 every **#pragma options enum=** directive that you put in your program, it is good practice to have a corresponding **#pragma options=reset** as well. This is the only way to prevent one file from potentially changing the **enum=** setting of another file that **#includes** it. It is good practice to specify **#pragma options enum=reset** at the end of any file that contains **#pragma options enum=** directives.

The table below shows the priority for selecting a predefined type. It 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**).

**Priority of Choosing Predefined enum Types**

| Priority    | Variable             | Constant      | Range (inclusive)                               | Size (bytes) |
|-------------|----------------------|---------------|-------------------------------------------------|--------------|
| 1 (highest) | unsigned char        | int           | 0 to 255                                        | 1            |
| 2           | signed char          | int           | -(127 + 1) to 127                               | 1            |
| 3           | unsigned short       | int           | 0 to 65,535                                     | 2            |
| 4           | short (signed short) | int           | -(32767 + 1) to 32767                           | 2            |
| 5           | unsigned int         | unsigned int  | 0 to 4,294,967,295                              | 4            |
| 6           | int (signed int)     | int           | -(2,147,483,647 + 1) to 2,147,483,647           | 4            |
| 7           | unsigned long        | unsigned long | 0 to 2 <sup>64</sup>                            | 8 (see Note) |
| 8 (lowest)  | signed long          | signed long   | -( 2 <sup>63</sup> ) to ( 2 <sup>63</sup> - 1 ) | 8 (see Note) |

**Note:** Long enum types are valid only in 64-bit compiler mode.

- When you specify **#pragma options enum=small**, the option stays in effect until it is explicitly turned off with a **#pragma options enum=int** or **#pragma options enum=reset** directive.
- If you compile the file using the **-qenum=int** option on the command line, the first **#pragma options=small** directive encountered in the source file will override it.
- If you specify **-qenum=small** on the command line, it is turned off by the first **#pragma options enum=int** directive found in the source code.
- 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 /* cannot be within a declaration */
  c
} e_var;
```

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 /* larger than maximum int */
             } e_var;
```

The **enum** constant range does not fit within the range of an **int (signed int)**.

```
#pragma options enum=small
enum e_tag { a=0,
             b=4294967296 /* larger than maximum int */
             } e_var;
```

The **enum** constant range does not fit within the range of an **unsigned int**.

```
#pragma options enum=small
enum e_tag { a=-1,
             b=2147483647, /* max int */
             c /* larger than maximum int */
             } e_var;
```

The **enum** constant range does not fit within the range of an **int (signed int)**.

The **#pragma options** keywords are **ENUM=SMALL**, to specify minimum-sized **ENUMS**; **ENUM=INT**, to disable minimum-sized enums; and **ENUM=RESET**, to reset the enum mapping rule to the rule that was in effect before the current mapping rule. If no previous enum mapping rule was specified in the file, the rule specified when the compiler was invoked is used.

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:

```
/*
 * 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
```

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)** has a higher priority, it will be used to represent the **enum**.

4. If you compile a file `myprogram.c` using the command:

```
xlc myprogram.c -qenum=small
```

assuming file `myprogram.c` does not contain **#pragma options=int** statements, all **enum** variables within your source file will occupy the minimum amount of storage.

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:

```
xlc yourfile.c
```

only the enum variable **first\_order** will be minimum-sized (that is, enum variable **first\_order** will only occupy 1 byte of storage). The other two enum variables **test\_enum** and **listening\_type** will be of type **int** and occupy 4 bytes of storage.

### Related Reading

“Compiler Options and Their Defaults” on page 218

## extchk

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | noextchk      | EXTCHK          |

### Syntax

```
-qextchk | -qnoextchk
EXTCHK | NOEXTCHK
```

### Purpose

Generates bind-time type checking information and checks for compile-time consistency.

### Notes

**-qextchk** checks for consistency at compile time and detects mismatches across compilation units at link time.

**-qextchk** 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:

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

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

## f

| Option Type  | Default Value | #pragma options |
|--------------|---------------|-----------------|
| <i>-flag</i> | -             | -               |

### Syntax

```
-f filename
```

### Purpose

Linkage editor (**ld** command) option only. Passes to the linkage editor the *filename* of a file containing a list of input files to be processed

### Notes

Each line in *filename* is treated as if it were listed separately on the **ld** linkage editor command line. Lines in this file can contain the following shell pattern characters to designate multiple object files:

- \* asterisk
- [ left bracket
- ] right bracket
- ? question mark

For more information on the **-f** compiler option, refer to the **ld** command in the *AIX Commands Reference*.

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Characteristics” on page 226

## F

| Option Type  | Default Value | #pragma options |
|--------------|---------------|-----------------|
| <i>-flag</i> | -             | -               |

### Syntax

```
-Fconfig_file:stanza | -Fconfig_file | -F:stanza
```

### Purpose

Names an alternative configuration file for **xlc**.

### Notes

*config\_file* Specifies the configuration of your system to the compiler.

*stanza* Is the name of the command used to invoke the compiler. This directs the compiler to the *config\_file* under *stanza* for the description of the compiler environment.

This suboption is not required.

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 “Specifying Compiler Options in a Configuration File” on page 13.

Options specified with **-W** option override options in the **-Fconfig\_file** configuration file. The **-B**, **-t**, and **-W** options override the **-F** option.

### Example

To compile *myprogram.c* using a configuration file **/usr/tmp/myvac.cfg** with an **xlc** stanza, enter:

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

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Characteristics” on page 226

“etc/vac.cfg - Default Configuration File” on page 474

“B” on page 239

“t” on page 329

“W” on page 336

## fdpr

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | nofdpr        | -               |

### Syntax

```
-qfdpr | -qnofdpr
```

### Purpose

Collects information about your program for use with the AIX **fdpr** (Feedback Directed Program Restructuring) performance-tuning utility.

### 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 Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229



## flag

| Option Type     | Default Value | #pragma options                  |
|-----------------|---------------|----------------------------------|
| <b>-qoption</b> | flag=i:i      | FLAG= <i>severity1:severity2</i> |

### Syntax

`-qflag=severity1:severity2 FLAG=severity1:severity2`

### Purpose

Specifies the minimum severity level of diagnostic messages to be reported in a listing and displayed on a terminal.

### Notes

*severity1*      Message level reported in listing  
*severity2*      Message level reported on terminal

You must specify a level for both *severity1* and *severity2*.

Diagnostic messages have the following severity levels:

**i**      Informational  
**w**      Warning  
**e**      Error  
**s**      Severe Error  
**u**      Unrecoverable Error

Specifying informational messages 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, enter:

```
xlc myprogram.c -qflag=I:E
```

### Related Reading

“Compiler Options and Their Defaults” on page 218  
“Options that Specify Compiler Output” on page 228  
“Message Severity Levels and Compiler Response” on page 20  
“info” on page 275  
“suppress” on page 328

## float

| Option Type     | Default Values                                                                                          | #pragma options |
|-----------------|---------------------------------------------------------------------------------------------------------|-----------------|
| <b>-qoption</b> | noemulate<br>nofltint<br>fold<br>nohsflt<br>nohssngl<br>maf<br>norndsngl<br>norm<br>norsqrt<br>nospnans | FLOAT           |

### Syntax:

-qfloat=*suboptions*  
 FLOAT=*suboptions*

**Purpose**

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

**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. See “Floating-Point Compiler Options” on page 27 before using this option.

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

|                                                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>-qfloat=emulate  <br/>-qfloat=noemulate</p> | <p>Emulates the floating-point instructions omitted by the PowerPC 403 processor. The default is <b>float=noemulate</b>.</p> <p>To emulate PowerPC 403 processor floating-point instructions, use <b>-qfloat=emulate</b>. 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 <b>-qfloat=emulate</b> with any of the following:</p> <ul style="list-style-type: none"> <li>• <b>-qarch=pwr, -qarch=pwr2, -qarch=pwrx</b></li> <li>• <b>-qlongdouble, -qldb128</b></li> <li>• xlc128 compiler invocation command</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <p>-qfloat=fltint  <br/>-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 <b>float=nofltint</b>, 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"> <li>• For <b>-O2</b>, the default is <b>-qfloat=nofltint</b>.</li> <li>• For <b>-O3</b>, the default is <b>-qfloat=fltint</b>.</li> </ul> <p>To include range checking in floating-point-to-integer conversions with the <b>-O3</b> option, specify <b>-qfloat=nofltint</b>.</p> <ul style="list-style-type: none"> <li>• <b>-qstrict</b> sets <b>-qfloat=nofltint</b></li> <li>• <b>-qnostrict</b> sets <b>-qfloat=fltint</b></li> </ul> <p>Changing the optimization level will not change the setting of the <b>fltint</b> option if <b>fltint</b> has already been specified.</p> <p>If <b>-qfloat=</b> options are explicitly set, the <b>-qstrict   -qnostrict</b> option will not override those settings. Otherwise, the default setting appearing last is used.</p> |
| <p>-qfloat=fold  <br/>-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 <b>-qfloat=fold</b> option replaces the obsolete <b>-qfold</b> option. Use <b>-qfloat=fold</b> in your new applications.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |

|                                               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-----------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>-qfloat=hsflt  <br/>-qfloat=nohsflt</p>    | <p>Speeds up calculations by enforcing the rounding of computed values to single precision before storing and on conversions from floating point to integer. <b>nohsflt</b> 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 <b>hsflt</b> option overrides the <b>rndsngl</b>, <b>nans</b>, and <b>spnans</b> options.</p> <p><b>Note:</b> The <b>hsflt</b> option 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 <b>-qfloat=hsflt</b> option replaces the obsolete <b>-qhsflt</b> option. Use <b>-qfloat=hsflt</b> in your new applications.</p> |
| <p>-qfloat=hssngl  <br/>-qfloat=nohssngl</p>  | <p>Specifies that single-precision expressions are rounded only when the results are stored into <b>float</b> memory locations. <b>nohssngl</b> specifies that single-precision expressions are rounded after expression evaluation. Using <b>hssngl</b> can improve runtime performance but is safer than using <b>-qfloat=hsflt</b>.</p> <p>The <b>-qfloat=hssngl</b> option replaces the obsolete <b>-qhssngl</b> option. Use <b>-qfloat=hssngl</b> in your new applications.</p>                                                                                                                                                                                                                                                                                                                                                                                   |
| <p>-qfloat=maf  <br/>-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 <b>-qfloat=maf</b> option replaces the obsolete <b>-qmaf</b> option. Use <b>-qfloat=maf</b> in your new applications.</p>                                                                                                                                                                                                                                                                                                                                                                            |
| <p>-qfloat=nans  <br/>-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 <b>nonans</b> specifies that this conversion need not be detected. <b>-qfloat=nans</b> is required for full compliance to the IEEE 754 standard.</p> <p>The <b>hsflt</b> option overrides the <b>nans</b> option.</p> <p>When used with the <b>-qflttrap</b> or <b>-qflttrap=invalid</b> option, the compiler detects invalid operation exceptions in comparison operations that occur when one of the operands is a signalling NaN.</p> <p>The <b>-qfloat=nans</b> option replaces the obsolete <b>-qfloat=spnans</b> option and the <b>-qspnans</b> option. Use <b>-qfloat=nans</b> in your new applications.</p>                                                                                          |
| <p>qfloat=rndsngl  <br/>-qfloat=norndsngl</p> | <p>Specifies that the result of each single-precision (<b>float</b>) operation is to be rounded to single precision. <b>-qfloat=norndsngl</b> 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 <b>hsflt</b> option overrides the <b>rndsngl</b> option.</p> <p>The <b>-qfloat=rndsngl</b> option replaces the obsolete <b>-qrndsngl</b> option. Use <b>-qfloat=rndsngl</b> in your new applications.</p>                                                                                                                                                                                                                                                                      |
| <p>-qfloat=rrm  <br/>-qfloat=norm</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><b>-qfloat=rrm</b> 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 <b>-qfloat=rrm</b> option replaces the obsolete <b>-qrrm</b> option. Use <b>-qfloat=rrm</b> in your new applications.</p>                                                                                                                                                                                                                                                              |

|                                                                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><code>-qfloat=rsqrt</code>  <br/><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"> <li>• For <b>-O2</b>, the default is <b>-qfloat=norsqrt</b>.</li> <li>• For <b>-O3</b>, the default is <b>-qfloat=rsqrt</b>. Use <b>-qfloat=norsqrt</b> to override this default.</li> <li>• <b>-qstrict</b> sets <b>-qfloat=norsqrt</b>.</li> <li>• <b>-qnostrict</b> sets <b>-qfloat=rsqrt</b>. (Note that <b>-qfloat=rsqrt</b> means that <b>errno</b> will <i>not</i> be set for any <b>sqrt</b> function calls.)</li> <li>• <b>-qfloat=rsqrt</b> has no effect when <b>-qarch=pwr2</b> is also specified.</li> <li>• <b>-qfloat=rsqrt</b> has no effect unless <b>-qignerrno</b> is also specified.</li> </ul> <p>Changing the optimization level will not change the setting of the <b>rsqrt</b> option if <b>rsqrt</b> has already been specified.</p> <p>If <b>-qfloat=</b> options are explicitly set, the <b>-qstrict</b>   <b>-qnostrict</b> option will not override those settings. Otherwise, the default setting appearing last is used.</p> |
| <p><code>-qfloat=spnans</code>  <br/><code>-qfloat=nospnans</code></p> | <p>Generates extra instructions to detect signalling NaN on conversion from single precision to double precision. The option <b>nospnans</b> specifies that this conversion need not be detected.</p> <p>The <b>hsflt</b> option overrides the <b>spnans</b> option.</p> <p>The <b>-qfloat=nans</b> option replaces the obsolete <b>-qfloat=spnans</b> and <b>-qspnans</b> options. Use <b>-qfloat=nans</b> in your new applications.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

### Example

To compile myprogram.c so that range checking occurs and multiply-add instructions are not generated, enter:

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

### Related Reading

- “Floating-Point Compiler Options” on page 27
- “Compiler Options and Their Defaults” on page 218
- “Options that Specify the Compiler Object Code Produced” on page 229
- “O, optimize” on page 302
- “arch” on page 237
- “fold” on page 265
- “hsflt” on page 271
- “hssngl” on page 272
- “ldbl128, longdouble” on page 289
- “maf” on page 297
- “rndsngl” on page 316
- “rrm” on page 318
- “spnans” on page 323
- “strict” on page 326

## flttrap

| Option Type           | Default Value | #pragma options |
|-----------------------|---------------|-----------------|
| <code>-qoption</code> | noflttrap     | FLTTRAP         |

### Syntax:

```
-qflttrap | -qflttrap=suboptions | -qnoflttrap  
FLTTRAP | FLTTRAP=suboptions | NOFLTTRAP
```

### Purpose

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

## Notes

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

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

The **fltrap** option has the following *suboptions*:

|            |                                                                                                                                                                               |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 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.        |

Specifying the **fltrap** option with no suboptions is equivalent to setting **-qfltrap=ov:und:zero:inv:inex**. The exceptions are not automatically enabled, and all floating-point operations are checked to provide precise exception-location information.

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 -qfltrap=overflow:underflow:zerodivide:enable
```

## Related Reading

- “Floating-Point Compiler Options” on page 27
- “Compiler Options and Their Defaults” on page 218
- “O, optimize” on page 302
- “float” on page 261
- “O, optimize” on page 302

## fold

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | fold          | FOLD            |

## Syntax:

```
-qfold | -qnofold  
FOLD | NOFOLD
```

### Purpose

Specifies that constant floating-point expressions are to be evaluated at compile time.

### Notes

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

### Related Reading

“Floating-Point Compiler Options” on page 27

“Compiler Options and Their Defaults” on page 218

“float” on page 261

## fullpath

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <i>-qoption</i> | nofullpath    | -               |

### Syntax

`-qfullpath | -qnofullpath`

### Purpose

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

### 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. Using **-qfullpath** would locate the file successfully.

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Debugging Features” on page 227

“g” on page 267

## G

| Option Type  | Default Value | #pragma options |
|--------------|---------------|-----------------|
| <i>-flag</i> | -             | -               |

### Syntax

This is a linkage editor (**ld**) option. Refer to *AIX Version 4 Commands Reference* for a description of **ld** command usage and syntax.

### Purpose

Tells the linkage editor to create a dynamic library.

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

## g

| Option Type  | Default Value | #pragma options |
|--------------|---------------|-----------------|
| <i>-flag</i> | -             | -               |

### Syntax

-g

### Purpose

Generates information used by debugging tools such as the **xldb** graphical debugger.

### 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.

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.

### Example

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

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

To compile myprogram.c to produce an executable program **testing\_all** containing additional information about unreferenced symbols so you can debug it, enter:

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

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Debugging Features” on page 227

“O, optimize” on page 302

“Q” on page 314

“dbxextra” on page 252

“fullpath” on page 266

“ipa” on page 279

“linedebug” on page 291

## genpcomp

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <i>-qoption</i> | nogenpcomp    | -               |

## Syntax

`-qgenpcomp` | `-qgenpcomp=directory` | `-qnoqgenpcomp`

## Purpose

Generates a precompiled version of any header file for which the original source file is used. This may help improve compile time when you use the **-qusepcomp** option.

## Notes

|                                   |                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-----------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-qgenpcomp</code>           | Generates a precompiled header file called <b>csetc.pch</b> , and saves it to the current directory.                                                                                                                                                                                                                                                                                                                   |
| <code>-qgenpcomp=directory</code> | Generates a precompiled header file. <ul style="list-style-type: none"><li>• If <i>directory</i> is the name of an existing directory, the precompiled header file is named <b>csetc.pch</b> and saved to that named <i>directory</i>.</li><li>• If a directory with the name <i>directory</i> does not exist, the precompiled header file is named <i>directory</i>, and is saved to the current directory.</li></ul> |
| <code>-qnoqgenpcomp</code>        | Does not generate precompiled header files.                                                                                                                                                                                                                                                                                                                                                                            |

**-qgenpcomp** and **-qusepcomp** will be ignored if they are both specified along with the **-a** or **-ae** options. Without the **-qusepcomp** option, **-qgenpcomp** is accepted in all cases.

## Example

To compile `myprogram.c` and generate a precompiled header file for any files that have changed since the last compilation, or for any files that do not have precompiled header files, and then place them in the directory **/headers**, enter:

```
xlc myprogram.c -qgenpcomp=/headers
```

The new precompiled header is called **csetc.pch**.

## Related Reading

“Creating and Using Precompiled Headers” on page 35

“Compiler Options and Their Defaults” on page 218

“usepcomp” on page 335

## genproto

| Option Type           | Default Value | #pragma options |
|-----------------------|---------------|-----------------|
| <code>-qoption</code> | nogenproto    | -               |

## Syntax

`-qgenproto` | `-qgenproto=parmnames` | `-qnoqgenproto`

## Purpose

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

## 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 Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Characteristics” on page 226

## halt

| Option Type           | Default Value | #pragma options       |
|-----------------------|---------------|-----------------------|
| <code>-qoption</code> | halt=s        | HALT= <i>severity</i> |

### Syntax

```
-qhalt=severity  
HALT=severity
```

### Purpose

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

### Notes

*severity* is one of:

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

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 Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Output” on page 228

“Message Severity Levels and Compiler Response” on page 20

“flag” on page 261

“maxerr” on page 299

## heapdebug

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <i>-qoption</i> | noheapdebug   | val             |

### Syntax

```
-qheapdebug
```

### Purpose

Enables debug versions of memory management functions.

### Notes

The **-qheapdebug** options specifies that the debug versions of memory management functions (**\_debug\_calloc**, **\_debug\_malloc**, **new**, etc.) be used in place of regular memory management functions. This option defines the **\_\_DEBUG\_ALLOC\_\_** macro.

When you specify **-qheapdebug**, the compiler generates additional code at the beginning of every function that preinitializes the local variables for the function. This makes it easier to find uninitialized local variables.

By default, the compiler uses the regular memory management functions (**calloc**, **malloc**, **new**, etc.) and does not preinitialize their local storage.

### Example

To compile myprogram.c with the debug versions of memory management functions, enter:

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

## Related Reading

- “Debugging Memory Heaps” on page 183
- “Memory Management Functions” on page 179
- “Managing Memory with Multiple Heaps” on page 182
- “Debugging Programs with Heap Memory” on page 195
- “Compiler Options and Their Defaults” on page 218
- “Options that Specify Debugging Features” on page 227
- “\_debug\_malloc - Allocate and Initialize Memory” on page 407
- “\_debug\_free - Free Allocated Memory” on page 408
- “\_debug\_heapmin - Free Unused Memory in the Default Heap” on page 410
- “\_debug\_malloc - Allocate Memory” on page 412
- “\_debug\_memcpy - Copy Bytes” on page 413
- “\_debug\_memmove - Copy Bytes” on page 415
- “\_debug\_memset - Set Bytes to Value” on page 416
- “\_debug\_realloc - Reallocate Memory Block” on page 417
- “\_debug\_strcat - Concatenate Strings” on page 419
- “\_debug\_strcpy - Copy Strings” on page 421
- “\_debug\_strncat - Concatenate Strings” on page 422
- “\_debug\_strncpy - Copy Strings” on page 423
- “\_debug\_strnset - Set Characters in String” on page 425
- “\_debug\_strset - Set Characters in String” on page 426
- “\_debug\_ucalloc - Reserve and Initialize Memory from User Heap” on page 428
- “\_debug\_uheapmin - Free Unused Memory in User Heap” on page 430
- “\_debug\_umalloc - Reserve Memory Blocks from User Heap” on page 431

## hsflt

| Option Type | Default Value | #pragma options |
|-------------|---------------|-----------------|
| -qoption    | nohsflt       | HSFLT           |

### Syntax:

```
-qhsflt | -qnohsflt  
HSFLT | NOHSFLT
```

### 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.

### Notes

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

The **hsflt** 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. See “Floating-Point Compiler Options” on page 27 before you use the **-qhsflt** option.

## Related Reading

- “Floating-Point Compiler Options” on page 27
- “Compiler Options and Their Defaults” on page 218
- “float” on page 261
- “rndsngl” on page 316
- “spnans” on page 323

# hssngl

| Option Type           | Default Value | #pragma options |
|-----------------------|---------------|-----------------|
| <code>-qoption</code> | nohssngl      | HSSNGL          |

## Syntax

```
-qhssngl | -qnohssngl  
HSSNGL | NOHSSNGL
```

## Purpose

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 valuation. Using **hssngl** can improve run-time performance.

## Notes

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

## Related Reading

"Floating-Point Compiler Options" on page 27  
"Compiler Options and Their Defaults" on page 218  
"float" on page 261

# I

| Option Type        | Default Value | #pragma options |
|--------------------|---------------|-----------------|
| <code>-flag</code> | -             | -               |

## Syntax

```
-Idirectory
```

## Purpose

Specifies an additional search path if the file name in the **#include** directive is not specified using its absolute path name.

## 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.

The normal search order 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 directory, **/usr/include**.

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.

When all specified directories have been searched, the directories on the standard list for **#include** files are searched. The directories on the standard list differ for the two versions of the **#include** directive. See "Directory Search Sequence for Include Files Using Relative Path Names" on page 178 for more information about searching directories.

The **-Idirectory** 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.

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 Reading

“Compiler Options and Their Defaults” on page 218

“idirfirst” Compiler Option

## idirfirst

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | noidirfirst   | IDIRFirst       |

### Syntax

```
-qidirfirst | -qnoidirfirst  
IDIRFIRST | NOIDIRFIRST
```

### Purpose

Specifies the search order for files included with the **#include** “*file\_name*” directive.

### Notes

Use **-qidirfirst** with the **-Idirectory** 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 **-Idirectory** option.
3. Search the standard include directory, `/usr/include`.

With **-qidirfirst**, the directories specified with the **-Idirectory** 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” on page 178.

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 Reading

“Compiler Options and Their Defaults” on page 218  
“l” on page 272  
“stdinc” on page 325

## ignerrno

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <i>-qoption</i> | noignerrno    | -               |

### Syntax

*-qignerrno* | *-qignerrno*

### Purpose

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

### Notes

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

### Related Reading

“Compiler Options and Their Defaults” on page 218  
“Options that Specify the Compiler Object Code Produced” on page 229

## ignprag

| Option Type     | Default Value | #pragma options           |
|-----------------|---------------|---------------------------|
| <i>-qoption</i> | -             | IGNPRAG= <i>suboption</i> |

### Syntax

*-qignprag=suboption*    *IGNPRAG=suboption*

### Purpose

Instructs the compiler to ignore certain pragmas.

### Notes

Suboptions are:

|                 |                                                                                                                                                   |
|-----------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>all</i>      | Equivalent to selecting all options described below.                                                                                              |
| <i>disjoint</i> | Ignores all <b>#pragma disjoint</b> directives in the source file.                                                                                |
| <i>isolated</i> | Ignores all <b>#pragma isolated_call</b> directives in the source file.                                                                           |
| <i>ibm</i>      | Ignores all IBM parallel processing directives in the source file, such as <b>#pragma ibm parallel_loop</b> , <b>#pragma ibm schedule</b> , etc.. |
| <i>omp</i>      | Ignores all OpenMP parallel processing directives in the source file, such as <b>#pragma omp parallel</b> , <b>#pragma omp critical</b> , etc..   |

The **ignprag** 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.

This option is also useful for disabling parallel processing directives to ensure that a program works correctly in both sequential and parallel mode.

## Examples

1. To compile myprogram.c and ignore any **#pragma isolated** directives, enter:

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

2. To compile myprogram.c and ignore all parallel processing pragmas, enter:

```
xlc myprogram.c -qignprag=ibm:omp
```

## Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Debugging Features” on page 227

“Example of the #pragma disjoint Preprocessor Directive” on page 367

“#pragma isolated\_call Preprocessor Directive” on page 371

“#pragma Preprocessor Directives for Parallel Processing” on page 381

“O, optimize” on page 302

## info

| Option Type           | Default Value | #pragma options |
|-----------------------|---------------|-----------------|
| <code>-qoption</code> | noinfo        | INFO            |

## Syntax

```
-qinfo | -qinfo=all | -qinfo=suboption[:suboption ...] | -qnoinfo  
INFO | INFO=ALL | INFO=suboption[:suboption ...] | INFO=RESET | NOINFO
```

## Purpose

Produces informational messages.

## Notes

Specifying **-qinfo** or **-qinfo=all** turns on 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 a particular section of program code, and **#pragma options info=reset** to return to your initial **-qinfo** settings.

Available *suboptions* for **-qinfo** compiler option are:

| Suboptions | Description                                                                                                                                               |
|------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| all        | Turns on all diagnostic messages for all groups.                                                                                                          |
| private    | <b>Note:</b> The <b>-qinfo</b> and <b>-qinfo=all</b> forms of the option have the same effect.<br>Lists shared variables made private to a parallel loop. |
| reduction  | Lists all variables that are recognized as reduction variables inside a parallel loop.                                                                    |

## Suboptions

*group*

## Description

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

### **group** Type of messages returned

|            |                                                              |
|------------|--------------------------------------------------------------|
| <b>cmp</b> | Possible redundancies in unsigned comparisons                |
| <b>cnd</b> | Possible redundancies or problems in conditional expressions |
| <b>cns</b> | Operations involving constants                               |
| <b>cnv</b> | Conversions                                                  |
| <b>dc1</b> | Consistency of declarations                                  |
| <b>eff</b> | Statements with no effect                                    |
| <b>enu</b> | Consistency of enum variables                                |
| <b>ext</b> | Unused external definitions                                  |
| <b>gen</b> | General diagnostic messages                                  |
| <b>gnr</b> | Generation of temporary variables                            |
| <b>got</b> | Use of goto statements                                       |
| <b>ini</b> | Possible problems with initialization                        |
| <b>inl</b> | Functions not inlined                                        |
| <b>lan</b> | Language level effects                                       |
| <b>obs</b> | Obsolete features                                            |
| <b>ord</b> | Unspecified order of evaluation                              |
| <b>par</b> | Unused parameters                                            |
| <b>por</b> | Nonportable language constructs                              |
| <b>ppc</b> | Possible problems with using the preprocessor                |
| <b>ppt</b> | Trace of preprocessor actions                                |
| <b>pro</b> | Missing function prototypes                                  |
| <b>rea</b> | Code that cannot be reached                                  |
| <b>ret</b> | Consistency of return statements                             |
| <b>trd</b> | Possible truncation or loss of data or precision             |
| <b>tru</b> | Variable names truncated by the compiler                     |
| <b>uni</b> | Uninitialized variables                                      |
| <b>use</b> | Unused auto and static variables                             |
| <b>vft</b> | Generation of virtual function tables                        |

## Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Output” on page 228

“#pragma info Preprocessor Directive” on page 370

## initauto

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | noinitauto    | INITAuto        |



## Syntax

```
-qinitauto=hex_value | -qnoinitauto  
INITAUTO=hex_value | NOINITAUTO
```

## Purpose

Initializes automatic storage to the two-digit hexadecimal byte value *hex\_value*. 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.

## Notes

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 Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

## inlglue

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | noinlglue     | INLGLUE         |

## Syntax

```
-qinlglue | -qnoinlglue  
INLGLUE | NOINLGLUE
```

## 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.

## Notes

*Glue code*, generated by the linker, is used for passing control between two external functions, or when you call functions through a pointer. Therefore the **-qinlglue** option only affects function calls through pointers or calls to an external compilation unit. For calls to an external function, you should specify that the function is imported by using, for example, the **-qprocimported** option.

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 Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Linkage Options” on page 230

“proclocal, procimported, procunknown” on page 312

“compact” on page 246

## inline

| Option Type     | Default Value     | #pragma options |
|-----------------|-------------------|-----------------|
| <b>-qoption</b> | <i>See below.</i> | -               |

## Syntax

-qinline | -qinline=*threshold* | -qinline=*names* | -qinline=*names* | -qinline=*limit* | -qnoinline

## 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.

## 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.

The C for AIX **\_\_inline**, **\_Inline**, and **\_\_inline** 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.

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

|                            |                                                                                                                                                                                                                                                                                                                                                                                                                              |
|----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -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 <b>-qinline</b> option. If <b>-qinline</b> is specified last, all functions are inlined.                                                                                                                                                                     |
| -qnoinline                 | Does not inline any functions. If <b>-qnoinline</b> is specified last, no functions are inlined.                                                                                                                                                                                                                                                                                                                             |
| -qinline= <i>threshold</i> | Sets a size limit on the functions to be inlined. The number of executable statements must be less than or equal to <i>threshold</i> for the function to be inlined. <i>threshold</i> must be a positive integer. The default value is 20. Specifying a threshold value of <b>0</b> causes no functions to be inlined except those functions marked with the <b>__inline</b> , <b>_Inline</b> , or <b>__inline</b> 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= <i>names</i> | The compiler does not inline functions listed by <i>names</i> . Separate each <i>name</i> with a colon (:). All other appropriate functions are inlined. The option implies <b>-qinline</b> . |
|------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

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.

`-qinline`

Is the same as `-Q`.

### 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 specify the **-O** option (to optimize your program) and the **-qinline** option (to inline functions), the compiler attempts to inline the functions you specify.

### Example

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

```
xlc myprogram.c -O -qnoinline
```

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

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

### Related Reading

“Program Optimization with the C for AIX Compiler” on page 23

“Using Inlined Components” on page 202

“Writing Optimized Program Source Code” on page 197

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

“\_Inline, \_inline, \_\_inline” on page 204, “\_Inline, \_inline, \_\_inline” on page 204, and “\_Inline,

\_inline, \_\_inline” on page 204

“g” on page 267

“O, optimize” on page 302

“Q” on page 314

## ipa

| Option Type           | Default Values |           | #pragma options |
|-----------------------|----------------|-----------|-----------------|
|                       | Compile-time   | Link-time |                 |
| <code>-qoption</code> | object         | noipa     | -               |

## Syntax

For compile-time use:

```
-qipa  
-qipa=object|noobject
```

For link-time use:

```
-qipa  
-qipa=suboption {, suboption}  
-qnoipa
```

## IPA at Compile Time

| <b>-qipa Compile-time Formats</b> | <b>Description</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|-----------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -qipa                             | Activates interprocedural analysis with the following <b>-qipa suboption</b> default: <ul style="list-style-type: none"><li>• <b>object</b></li></ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| -qipa=object                      | Specifies whether to include standard object code in the object files.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| -qipa=noobject                    | Specifying the <b>noobject</b> suboption can substantially reduce overall compile time by not generating object code during the first IPA phase.<br><br>If the <b>-S</b> compiler option is specified with <b>noobject</b> , <b>noobject</b> is ignored.<br><br>If compilation and linking are performed in the same step, and neither the <b>-S</b> nor any listing option is specified, <b>-qipa=noobject</b> is implied by default.<br><br>If any object file used in linking with <b>-qipa</b> was created with the <b>-qipa=noobject</b> 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 <b>-qipa</b> . |

## IPA at Link Time

| <b>-qipa Link-time Formats</b> | <b>Description</b>                                                                                                                                                                                                                                                                                                  |
|--------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -qnoipa                        | Deactivates interprocedural analysis.                                                                                                                                                                                                                                                                               |
| -qipa                          | Activates interprocedural analysis with the following <b>-qipa suboption</b> defaults: <ul style="list-style-type: none"><li>• <b>inline=auto</b></li><li>• <b>level=1</b></li><li>• <b>missing=unknown</b></li><li>• <b>partition=medium</b></li></ul>                                                             |
| <i>suboption</i>               | Suboptions can take any of the forms shown below. Separate multiple suboptions with commas.<br><br><b>exits=name{,name}</b><br>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[=*suboption*]**

Same as specifying the **-qinline** compiler option, with *suboption* being any valid **-qinline** suboption.

**inline=auto**

Enables automatic inlining only. The compiler still accepts user-specified functions as candidates for inlining.

**inline=noauto**

Disables automatic inlining only. The compiler still accepts user-specified functions as candidates for inlining.

**inline=name{,*name*}**

Specifies a comma-separated list of functions to try to inline, where functions are identified by *name*.

**noinline=name{,*name*}**

Specifies a comma-separated list of functions that must not be inlined, where functions are identified by *name*.

**inline=limit=num**

Changes the size limits that the **-Q** option uses to determine how much inline expansion to perform. This established limit is the size below which the calling procedure must remain. *number* 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=size**

Specifies the upper size limit of functions to be inlined, where *size* is a value as defined under **inline=limit**. This argument is implemented only when **inline=auto** is on.

**isolated=name,{*name*}**

Specifies a list of *isolated* functions that are not compiled with IPA. Neither isolated functions nor functions within their call chain can refer to global variables.

**level=n** Specifies the optimization level for interprocedural analysis. The default level is 1. Valid levels are as follows:

- 0** Does only minimal interprocedural analysis and optimization.
- 1** Turns on inlining, limited alias analysis, and limited call-site tailoring.
- 2** Performs full interprocedural data flow and alias analysis.

**list[=*name*/short|long]**

Specifies that a listing file be generated during the link phase. The listing file contains information about transformations and analyses performed by IPA, as well as an optional object listing generated by the back end for each partition. This option can also be used to specify the name of the listing file.

If listings have been requested (using either the **-qlist** or **-qipa=list** options), and *name* is not specified, the listing file name defaults to **a.lst**.

The **long** and **short** suboptions can be used to request more or less information in the listing file. The **short** suboption, which is the default, generates the Object File Map, Source File Map and Global Symbols Map sections of the listing. The **long** suboption causes the generation of all of the sections generated through the short suboption, as well as the Object Resolution Warnings, Object Reference Map, Inliner Report and Partition Map sections.

**lowfreq=name{,name}**

Specifies names of functions which are likely to be called infrequently. These will typically be error handling, trace, or initialization functions. The compiler may be able to make other parts of the program run faster by doing less optimization for calls to these functions.

**missing=attribute**

Specifies the interprocedural behavior of procedures that are not compiled with **-qipa** and are not explicitly named in an **unknown**, **safe**, **isolated**, or **pure** suboption.

The following attributes may be used to refine this information:

**safe** Functions which do not indirectly call a visible (not missing) function either through a direct call or through a function pointer.

**isolated**

Functions which do not directly reference global variables accessible to visible functions. Functions bound from shared libraries are assumed to be *isolated*.

**pure** Functions which are *safe* and *isolated* and which do not indirectly alter storage accessible to visible functions. *pure* functions also have no observable internal state.

**unknown**

The default setting. This option greatly restricts the amount of interprocedural optimization for calls to *unknown* functions. Specifies that the missing functions are not known to be *safe*, *isolated*, or *pure*.

**partition=size**

Specifies the size of each program partition created by IPA during pass 2. *Size* can be any of:

- small
- medium
- large
- *any positive integer value*

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 *size* 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.

**pure=name{,name}**

Specifies a list of *pure* functions that are not compiled with **-qipa**. Any function specified as *pure* must be *isolated* and *safe*, and must not alter the internal state nor have side-effects, defined as potentially altering any data visible to the caller.

**safe=name{,name}**

Specifies a list of *safe* functions that are not compiled with **-qipa**. *Safe* functions can modify global variables, but may not call functions compiled with **-qipa**.

**unknown=name{,name}**

Specifies a list of *unknown* functions that are not compiled with **-qipa**. Any function specified as *unknown* can make calls to other parts of the program compiled with **-qipa**, and modify global variables and dummy arguments.

### *filename*

Gives the name of a file which contains suboption information in a special format. The file format is the following:

```
# ... comment
attribute{, attribute} = name{, name}
missing = attribute{, attribute}
exits = name{, name}
lowfreq = name{, name}
inline [ = auto | = noauto ]
inline = name{, name} [ from name{, name}]
inline-threshold = unsigned_integer
inline-limit = unsigned_integer
list [ = file-name | short | long ]
noinline
noinline = name{, name} [ from name{, name}]
level = 0 | 1 | 2
partition = small | medium | large | unsigned_integer
```

where *attribute* is one of:

- exits
- lowfreq
- unknown
- safe
- isolated
- pure

## Purpose

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

## Notes

1. 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.
2. Specify the **-qipa** option on both the compile and link steps of the entire application, or as much of it as possible. You should compile at least the file containing **main**, or at least one of the entry points if compiling a library.
3. 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:
  - a. 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 it's position relative to other automatics. Do not compile such a function with IPA (and expect it to work).
  - b. Accessing either an 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.
4. 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.
5. 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.

6. 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.
7. 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. Perform preliminary performance analysis and tuning before compiling with the **-qipa** option, because the IPA analysis uses a two-pass mechanism that increases compile and link time. You can reduce some compile and link overhead by using the **-qipa=noobject** option.
2. Specify the **-qipa** option on both the compile and the link steps of the entire application, or as much of it as possible. Use suboptions to indicate assumptions to be made about parts of the program *not* compiled with **-qipa**. During compilation, the compiler stores interprocedural analysis information in the **.o** file. During linking, the **-qipa** option causes a complete recompilation of the entire application.

**Note:** If a Severe error occurs during compilation, **-qipa** returns RC=1 and terminates. Performance analysis also terminates.

### Example

To compile a set of files with interprocedural analysis, enter:

```
xlc -c -O3 *.c -qipa
xlc -o product *.o -qipa
```

Here is how you might compile the same set of files, improving the optimization of the second compilation, and the speed of the first compile step. Assume there exists two functions, *trace\_error* and *debug\_dump*, which are rarely executed.

```
xlc -c -O3 *.c -qipa=noobject
xlc -c -O3 *.o -qipa=lowfreq=trace_error,debug_dump
```

If a given compiler option is specified at both compile- and link-time with differing settings, the link-time option settings will generally prevail. In the example below, the **-O3** option used at link-time, along with other settings implied by **-O3**, overrides the **-O2** option used at compile-time.

```
xlc -c -O2 *.c -qipa=noobject
xlc -c -O3 *.o -qipa=lowfreq=trace_error,debug_dump
```

### Related Reading

- “Compiler Options and Their Defaults” on page 218
- “Options that Specify the Compiler Object Code Produced” on page 229
- “S” on page 319
- “inline” on page 277
- “list” on page 291

## isolated\_call

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | -             | ISOLATED_CALL   |

### Syntax

```
-qisolated_call=function_name
ISOLATED_CALL=function_name
```



## Purpose

Specifies functions in the source file that have no side effects.

## Notes

*function\_name* Is the name of a function that does not have side effects 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 (:).

Marking a function as isolated can improve the runtime performance of optimized code by indicating to the optimizer that external and static variables are not changed by the called function.

The **#pragma options** keyword **isolated\_call** must be specified at the top of the file, before the first C statement. You can use the **#pragma isolated\_call** directive at any point in your source file.

## Example

To compile myprogram.c, specifying that the functions **myfunction(int)** and **classfunction(double)** do not have side effects, enter:

```
xlc myprogram.c -qisolated_call=myfunction:classfunction
```

## Related Reading

“Compiler Options and Their Defaults” on page 218

“#pragma options Preprocessor Directive” on page 375

“#pragma isolated\_call Preprocessor Directive” on page 371

## L

| Option Type  | Default Value     | #pragma options |
|--------------|-------------------|-----------------|
| <i>-flag</i> | <i>See below.</i> | -               |

## Syntax

*-Ldirectory*

## Purpose

Searches the path *directory* for library files specified by the **-lkey** option.

## 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** is searched for the library **libspfiles.a**, enter:

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

### Related Reading

“Compiler Options and Their Defaults” on page 218  
“Options that Specify Linkage Options” on page 230  
“l”

## l

| Option Type  | Default Value     | #pragma options |
|--------------|-------------------|-----------------|
| <i>-flag</i> | <i>See below.</i> | -               |

### Syntax

*-lkey*

### Purpose

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

### Notes

The actual search path can be modified with the **-L***directory* option. See **-B**, **-brtl**, and **-bstatic**, **-bdynamic** for information on specifying the types of libraries that are searched (for static or dynamic linking).

### Default

The default is to search only the C library (**-lc**).

### Example

To compile *myprogram.c* and include the Task Library, **libtask.a**, and the Complex Mathematics Library, **libcomplex.a**, enter:

```
xlc myprogram.c -ltask -lcomplex
```

### Related Reading

“Compiler Options and Their Defaults” on page 218  
“Options that Specify Linkage Options” on page 230  
“B” on page 239  
“datalocal, dataimported” on page 251  
“bstatic, bdynamic” on page 241  
“L” on page 285  
“l”

## langlvl

| Option Type     | Default Value        | #pragma options          |
|-----------------|----------------------|--------------------------|
| <i>-qoption</i> | <i>langlvl=ansi*</i> | <i>LANGlvl=suboption</i> |

### Syntax

```
-qlanglvl=suboption[:suboption ...]  
LANGlvl=suboption[:suboption ...]
```

### Purpose

Selects the C language level used for compilation.

### Default

The default language level is **ansi** when you invoke the compiler using the **xlc** or **c89** command. The default language level is **extended** when you invoke the compiler using the **cc** command.

You can use either of the following preprocessor directive styles to specify the language level used when compiling your C source program:

```
#pragma options langlvl=suboption[:suboption ...]
#pragma langlvl(suboption)
```

The **pragma** directive must appear before any noncommentary lines in the source code.

## Notes

Language level suboptions can be specified using an abbreviation of the complete suboption name. In the list below, the minimum suboption specification is shown with uppercase characters. Suboption names can be entered using either uppercase or lowercase characters.

| Suboption | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ANSI      | Compilation conforms to the ANSI language level standard.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| SAAL2     | Compilation conforms to the SAA C Level 2 CPI language level definition, with some exceptions.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| SAA       | Compilation conforms to the current SAA C CPI language level definition. This is currently SAA C Level 2.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| EXTended  | Provides compatibility with the RT compiler and <b>classic</b> language levels.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| CLAssic   | Allows the compilation of non-ANSI language level programs, and conforms closely to the K&R level preprocessor.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| NOUCS     | The default is NOUCS. This suboption can be used together with other <b>-qlanglvl</b> suboptions.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| UCS       | <p>With option <b>-qlanglvl=ucs</b>, you can use universal character names in form of <code>\unnnn</code> or <code>\Unnnnnnnn</code> as defined in the C9X Final Draft International Standard ISO/IEC 9899:1999.</p> <p>The universal character name <code>\unnnn</code> designates a character whose four-digit short identifier is <code>nnnn</code>. The universal character name <code>\Unnnnnnnn</code> designates a character whose eight-digit short identifier is <code>nnnnnnnn</code>. Short identifiers of characters are specified by ISO/IEC 10646. A four-digit identifier of <code>nnnn</code> is identical to an eight-digit short identifier of <code>0000nnnn</code>.</p> <p>Universal character names may be used in identifiers, character constants, and string literals to designate characters that are not in the basic character set. A universal character name shall not specify a character whose short identifier is:</p> <ul style="list-style-type: none"><li>• less than 00A0 except 0024 (\$), 0040 (@), and 0060 ('), or,</li><li>• in the range D800 through DFFF inclusive.</li></ul> |

If more than one language level is specified, the later option will override earlier options. For example, specifying

```
-qlanglvl=ansi:extended
```

will result in the compiler using the extended language level. The exceptions to this rule are the **nocucs** and **ucs** suboptions, which do not override and are not overridden by other **-qlanglvl** suboptions.

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 #dfine 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 **-qplusplus** is specified.)

**#line** The spelling of all tokens which are not part of the line number form the new file name. These tokens need not be string literals.

**#error** Not recognized in **classic** mode.

**#define**

A valid macro parameter list consists of zero or more identifiers each separated by commas. The commas are ignored and the parameter list is constructed as if they were not specified. The parameter names need not be unique. If there is a conflict, the last name specified is recognized.

For an invalid parameter list, a warning is issued. If a macro name is redefined with a new definition, a warning will be issued and the new definition used.

**#undef** When there are extra tokens, no diagnostic message is generated.

#### Macro expansion

- When the number of arguments on a macro invocation does not match the number of parameters, a warning is issued.
- If the ( token is present after the macro name of a function-like macro, it is treated as too few arguments (as above) and a warning is issued.
- Parameters are replaced in string literals and character literals.
- Examples:

```
#define M() 1
#define N(a) (a)
#define O(a,b) ((a) + (b))

M(); /* no error */
N(); /* empty argument */
O(); /* empty first argument
      and too few arguments */
```

#### Text Output

No text is generated to replace comments.

#### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Characteristics” on page 226

“cplusplus” on page 247

## ldbl128, longdouble

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | noldbl128     | LDL128          |

## Syntax

-qldb128 | -qnoldb128 | -qlongdouble | -qnoqlongdouble  
LDBL128 | NOLDBL128 | LONGDOUBLE | NOLONGDOUBLE

## Purpose

Increases the size of **long double** type from 64 bits to 128 bits.

## Notes

The **-qlongdouble** option is the same as the **-qldb128** 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 (**xlc128** or **cc128**). 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                       | N/A                             | libC128.a           | N/A                             |
| libCns.a                     | -lCns                           | libC128ns.a         | -lC128ns                        |
| libcomplex.a                 | -lcomplex                       | libcomplex128.a     | -lcomplex128                    |

Linking without the 128-bit versions of the libraries when your program uses 128-bit **long doubles** (for example, if you specify **-qldb128** alone) may produce unpredictable results.

The **-qldb128** option defines **\_\_LONGDOUBLE128**.

The **#pragma options** directive must appear before the first 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:

```
xlc myprogram.c -qldb128 -lC128
```

or:

```
xlc128 myprogram.c
```

For a description of the 128-bit long double, refer to “Implementation Dependency - Floating Point Types (F.3.6)” on page 451.

## Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

“l” on page 286

## libansi

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | nolibansi     | -               |

## Syntax

-qlibansi | -qnoqlibansi

## Purpose

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

## 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 Reading

“Compiler Options and Their Defaults” on page 218

## linedebug

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <i>-qoption</i> | nolinedebug   | -               |

## Syntax

`-qLINEDebug` | `-qNOLINEDebug`

## Purpose

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

## 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 Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Debugging Features” on page 227

“g” on page 267

“O, optimize” on page 302

“Q” on page 314

## list

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <i>-qoption</i> | nolist        | LIST            |

## Syntax

`-qlist` | `-qnolist`  
`LIST` | `NOLIST`

## Purpose

Produces a compiler listing that includes an object listing.

## Notes

Options that are not defaults appear in all listings, even if **nolist** is specified. The **noprint** option overrides this option.

## Example

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

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

## Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Output” on page 228

## listopt

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <i>-qoption</i> | nolistopt     | -               |

## Syntax

```
-qlistopt | -qnolistopt
```

## 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.

## Example

To compile myprogram.c to produce a compiler listing that shows all options in effect, enter:

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

## Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Output” on page 228

## longlit

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <i>-qoption</i> | nolonglit     | -               |

## Syntax

```
-qlonglit | -qnolonglit
```

## Purpose

Changes implicit type selection in 64-bit mode to use larger data types where possible.

## Notes

This feature provides the same effect as suffixing all integer constants with **I** or **L**.

This option may be useful in porting to 64-bit situations where a **signed long** result is expected instead of **unsigned int** in expressions that contain literals. For example:



```

unsigned int ui = 0;
long l;
l = ui - 1;

```

In 32-bit mode, **l** will be equal to -1. In 64-bit mode, the value of **l** becomes `UINT_MAX`. Forcing 1 into type **signed long** will provide the desired result.

Use this option with extreme caution as it implicitly changes the type of all unsuffixed integer constants that would otherwise have type **int** or **unsigned int**.

The following table shows implicit type selections performed by the compiler with and without the **longlit** option in effect.

|                                | Default 64-bit mode                                        | -qlonglit option enabled     |
|--------------------------------|------------------------------------------------------------|------------------------------|
| <b>unsuffixed decimal</b>      | signed int<br>signed long<br>unsigned long                 | signed long<br>unsigned long |
| <b>unsuffixed octal or hex</b> | signed int<br>unsigned int<br>signed long<br>unsigned long | signed long<br>unsigned long |
| <b>suffixed by u or U</b>      | unsigned int<br>unsigned long                              | unsigned long                |
| <b>suffixed by l or L</b>      | signed long<br>unsigned long                               | signed long<br>unsigned long |
| <b>suffixed by ul or UL</b>    | unsigned long                                              | unsigned long                |

#### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

## longlong

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | longlong*     | -               |

#### Syntax

```
-qlonglong | -qolonglong
```

#### Purpose

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

#### Default

The default with **xlc**, and **cc** is **-qlonglong**, which defines `_LONG_LONG` (**long long** types will work in C programs). The default with **c89** is **-qolonglong** (**long long** types are ignored).

#### Example

To compile `myprogram.c` so that **long long ints** are not allowed, enter:

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

#### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

## M

| Option Type  | Default Value | #pragma options |
|--------------|---------------|-----------------|
| <i>-flag</i> | -             | -               |

### Syntax

`-M`

### Purpose

Creates an output file that contains targets suitable for inclusion in a description file for the AIX **make** command.

### 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 *AIX Version 4 Commands Reference*.

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:

```
xlc -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:

```
xlc -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:

```
xlc -M -c t.c -o /tmp/t.o
```

places the **.u** output file in **/tmp/t.u**.

### Format of the Output File

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.cfile_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 178. (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 Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Preprocessor Options” on page 228

“o” on page 305

“makedep” on page 298

## ma

| Option Type  | Default Value | #pragma options |
|--------------|---------------|-----------------|
| <i>-flag</i> | -             | -               |

### Syntax

-ma

### Purpose

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

### 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.

### Example

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

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

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

## macpstr

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <i>-qoption</i> | nomacpstr     | MACPSTR         |

### Syntax

```
-qmacpstr | -qnomacpstr  
MACPSTR | NOMACPSTR
```

### Purpose

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

### 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 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 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 Reading

“Compiler Options and Their Defaults” on page 218

“mbs, dbs” on page 301

## maf

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | maf           | MAF             |

### Syntax

```
-qmaf | -qnoma f  
MAF | NOMAF
```

### Purpose

Specifies whether floating-point multiply-add instructions are to be generated. This option affects the precision of floating-point intermediate results. Before using this option, see “Floating-Point Compiler Options” on page 27 for more information about floating-point operations.

## Notes

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

## Related Reading

“Floating-Point Compiler Options” on page 27

“Compiler Options and Their Defaults” on page 218

“float” on page 261

## makedep

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | -             | -               |

## Syntax

`-qmakedep`

## Purpose

Creates an output file that contains targets suitable for inclusion in a description file for the AIX **make** command.

## 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 *AIX Version 4 Commands Reference*.

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:

```
xlc -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:

```
xlc -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:

```
xlc -qmakedep -c t.c -o /tmp/t.o
```

places the **.u** output file in **/tmp/t.u**.

## Format of the Output File

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.cfile_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 178. (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 Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Preprocessor Options” on page 228

“M” on page 294

“o” on page 305

## maxerr

| Option Type           | Default Value | #pragma options |
|-----------------------|---------------|-----------------|
| <code>-qoption</code> | nomaxerr      | -               |

### Syntax

```
-qmaxerr=num:[sev_level] | -qnomaxerr
```

### Purpose

Instructs the compiler to halt compilation when *num* errors of severity *sev\_level* or higher is reached.

### Notes

*num* must be an integer. *sev\_level* must be one of the following:

| sev_level | Description   |
|-----------|---------------|
| <b>i</b>  | Informational |
| <b>w</b>  | Warning       |
| <b>e</b>  | Error         |
| <b>s</b>  | Severe error  |

If no value is specified for *sev\_level*, 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.

Messages suppressed by the **-qsuppress** option are not counted.

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** and **-qsuppress** options.

### Examples

1. To stop compilation of `myprogram.c` when 10 warnings are encountered, enter the command:

```
xlc myprogram.c -qmaxerr=10:w
```

1. To stop compilation of `myprogram.c` when 5 severe errors are encountered, assuming that the current **-qhalt** option value is **S** (severe), enter the command:

```
xlc myprogram.c -qmaxerr=5
```

1. To stop compilation of `myprogram.c` when 3 informationals are encountered, enter the command:

```
xlc myprogram.c -qmaxerr=3:i
```

or:

```
xlc myprogram.c -qmaxerr=5:w qmaxerr=3 -qhalt=i
```

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Output” on page 228

“Message Severity Levels and Compiler Response” on page 20

“flag” on page 261

“halt” on page 269

“suppress” on page 328

## maxmem

| Option Type           | Default Value            | #pragma options |
|-----------------------|--------------------------|-----------------|
| <code>-qoption</code> | <code>maxmem=2048</code> | -               |

### Syntax

```
-qmaxmem=size
```

### 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.

### 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 **maxmem** 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.
- The option **-O3** implies **-qmaxmem=-1**.

The default is **-qmaxmem=2048**, which specifies a default memory size.

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 **4096** kilobytes, enter:

```
xlc myprogram.c -qmaxmem=4096
```



### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

“O, optimize” on page 302

## mbs, dbcs

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <i>-qoption</i> | nombcs        | DBCS            |

### Syntax:

`-qmbcs | -qdbc | -qnombcs | -qnodbc`  
`MBCS | DBCS | NOMBCS | NODBCS`

### Purpose

Use the **-qmbcs** option if your program contains multibyte characters. The **-qmbcs** option is equivalent to **-qdbc**.

### Notes

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

### Example

To compile `myprogram.c` if it contains multibyte characters, enter:

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

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Characteristics” on page 226

## noprint

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <i>-qoption</i> | -             | -               |

### Syntax

`-qnoprint`

### Purpose

Suppresses listings. **-qnoprint** overrides all of the listing-producing options, regardless of where they are specified.

### Notes

The default is not to 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:

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

## Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Output” on page 228

“att” on page 238

“list” on page 291

“listopt” on page 292

“source” on page 322

“xref” on page 339

## O, optimize

| Option Type                     | Default Value | #pragma options |
|---------------------------------|---------------|-----------------|
| <b>-qoption</b><br><i>-flag</i> | nooptimize    | -               |

## Syntax

```
-0 | -O2 | -O3 | -O4 | -qoptimize | -qoptimize=2 | -qoptimize=3 |  
-qoptimize=4 | -qoptimize=5 | -qnooptimize | -qoptimize=0  
OPTimize | OPTimize=2 | OPTimize=3 | OPTimize=4 | OPTimize=5 |  
NOOPTimize | OPTimize=0
```

## Purpose

Optimizes code at a choice of levels during compilation.

## 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 the **dbx** symbolic debug program. The debugging information produced may not be accurate.

The levels of optimization are:

**-qNOOPTimize** (Same as **-qOPTimize=0**.) Performs only quick local optimizations such as constant folding and elimination of local common subexpressions.

**-O, -qOPTimize** This setting implies **-qstrict\_induction** unless **-qnostrict\_induction** is explicitly specified. Performs optimizations that the compiler developers considered the best combination for compilation speed and runtime performance.

The optimizations may change from one product release to the next. If you need a specific level of optimization, specify the appropriate numeric value.

**-O2, -qOPTimize=2** This setting implies **-qnostrict\_induction** unless **-qstrict\_induction** is explicitly specified. Same as **-O**.

**-O3, -qOPTimize=3** Performs additional optimizations that are memory intensive, compile-time intensive, or both. These optimizations are performed in addition to those performed with only the **-O** option specified. They are recommended when the desire for runtime improvement outweighs the concern for minimizing compilation resources.

This level is the compiler's highest and most aggressive level of optimization. **-O3** performs optimizations that have the potential to slightly alter the semantics of your program. It also applies the **-O2** level of optimization with unbounded time and memory. The compiler guards against these optimizations at **-O2**.

You can use the **-qstrict** option with **-O3** to turn off the aggressive optimizations that might change the semantics of a program. **-qstrict** combined with **-O3** invokes all the optimizations performed at **-O2** as well as further loop optimizations. Note that the **-qstrict** compiler option must appear after the **-O3** option, otherwise it is ignored.

The aggressive optimizations performed when you specify **-O3** are:

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

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

For example, a loop-invariant floating-point computation that is found on some, but not all, paths through a loop will not be moved at **-O2** because the computation may cause an exception. At **-O3**, the compiler will move it because it is not certain to cause an exception. The same is true for motion of loads. Although a load through a pointer is never moved, loads off the static or stack base register are considered movable at **-O3**. Loads in general are not considered to be absolutely safe at **-O2** because a program can contain a declaration of a static array `a` of 10 elements and load `a[60000000003]`, which could cause a segmentation violation.

The same concepts apply to scheduling.

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

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

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

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

2. Conformance to IEEE rules are relaxed.

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

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

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

3. Floating-point expressions may be rewritten.

Computations such as  $a*b*c$  may be rewritten as  $a*c*b$  if, for example, an opportunity exists to get a common subexpression by such rearrangement. Replacing a divide with a multiply by the reciprocal is another example of reassociating floating-point computations.

## Notes

- **-qfloat=fltint:rsqrt** are on by default in **-O3**.
- Built-in functions do not change **errno** at **-O3**.
- Aggressive optimizations do *not* include the following floating-point suboptions: **-qfloat=hsflt**, **hssngl**, and **-qfloat=rndsngl**.
- The default **maxmem** value is **-1** at **-O3**.
- Refer to **-qflttrap** to see the behavior of the compiler when you specify **optimize** options with the **flttrap** option.
- The **-O3** option implies **-qnostrict**. You can use the **-qstrict** compiler option to turn resulting optimizations that can potentially change the semantics of a program. Reference to these compiler option must appear after the **-O3** option.
- The **-O3** compiler option implies **-qnostrict\_induction** unless **-qstrict\_induction** is explicitly specified.

**-O4, -qOPTimize=4** This option is the same as **-O3**, except that it also:

- Sets the **-qipa** option
- Sets the **-qarch** and **-qtune** options to the architecture of the compiling machine

**Note:** Later settings of **-O**, **-qcache**, **-qipa**, **-qarch**, and **-qtune** options will override the settings implied by the **-O4** option.

**-O5, -qOPTimize=5** This option is the same as **-O4**, except that it:

- Sets the **-qipa=level=2** option to perform full interprocedural data flow and alias analysis.

**Note:** Later settings of **-O**, **-qcache**, **-qipa**, **-qarch**, and **-qtune** options will override the settings implied by the **-O5** option.

## Example

To compile `myprogram.c` for maximum optimization, enter:

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

For an in-depth discussion of how to optimize and tune your programs, refer to the *Optimization and Tuning Guide for Fortran, C, and C++*.

## Related Reading

- “Program Optimization with the C for AIX Compiler” on page 23
- “Writing Optimized Program Source Code” on page 197
- “Minimizing the Size of Object Files” on page 36
- “Compiler Options and Their Defaults” on page 218
- “Options that Specify the Compiler Object Code Produced” on page 229
- “#pragma option\_override Preprocessor Directive” on page 374
- “g” on page 267
- “arch” on page 237
- “cache” on page 243
- “float” on page 261
- “flttrap” on page 264
- “ipa” on page 279
- “strict” on page 326
- “strict\_induction” on page 327
- “tune” on page 331

## O

| Option Type  | Default Value | #pragma options |
|--------------|---------------|-----------------|
| <i>-flag</i> | -             | -               |

## Syntax

`-o file_spec`

### 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.

### 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 following compiler invocation:

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

produces the object file **new.o** instead of **test.o** , and

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

produces the object file **new** instead of **a.out**

A *file\_spec* with a C source file suffix (`.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.

To use “**c**” on page 242 and `-o` together, you can only compile one source file at a time. If you specify both `-c` and `-ofile_spec`, and only one file is being compiled, the output is placed in *file\_spec*. If more than one source file name is listed in the compiler invocation, the compiler issues a warning message and ignores `-o`.

The “**E**” on page 253, “**P**” on page 307, and “**syntaxonly**” on page 327 options override the `-ofilename` option.

### Example

1. To compile `myprogram.c` so that the resulting file is called **myaccount**, assuming that no directory with name **myaccount** exists, enter:

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

If the directory **myaccount** does exist, the executable file produced by the compiler is placed in the **myaccount** directory.

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Linkage Options” on page 230

“**c**” on page 242

“**E**” on page 253

“**P**” on page 307

“**syntaxonly**” on page 327

### once

| Option Type           | Default Value | #pragma options |
|-----------------------|---------------|-----------------|
| <code>-qoption</code> | noonce        | ONCE            |

### Syntax

```
-qonce | -qnoonce  
ONCE | NOONCE
```

## Purpose

Avoids including a header file more than once even if it is specified in several of the files you are compiling.

## Notes

The compiler uses the full path name to determine if a file has already been included. No attempt is made to resolve `.` or `..` in the path name. `#include` statements that include `.` or `..` in the path statements may cause the same file to be included more than once.

The `#pragma options` keyword **ONCE** may appear anywhere in your code. It can be turned on and off by specifying **ONCE** and **NOONCE**, respectively.

## Important!

Do not use the `-qonce` option if *both* of the following conditions are true:

1. You include both `stdio.h` and `stdarg.h` (in that order) in your source files, and,
2. You are using the macro `va_list`. `va_list` must be defined twice to have any effect, and `-qonce` defeats this purpose.

## Example

The following example shows how the compiler resolves whether a file has already been included.

```
#include <stdio.h>          /* Found in /usr/include/stdio.h */
#include <stdio.h>          /* Already included */
#include </usr/include/stdio.h> /* Already included */
#include <./stdio.h>        /* Resolves to /usr/include/./stdio.h */
                           /* which is the same file, but this */
                           /* file will be included again. */
```

## Related Reading

“Compiler Options and Their Defaults” on page 218

# P

| Option Type        | Default Value | #pragma options |
|--------------------|---------------|-----------------|
| <code>-flag</code> | -             | -               |

## Syntax

`-P`

## Purpose

Preprocesses the 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`. The `-P` option calls the preprocessor directly as `/usr/vac/exe/xlCcpp`.

## 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.

`#linedirectives` 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 source files to produce an executable file.

#### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Output” on page 228

“C” on page 242

“c” on page 242

“E” on page 253

“o” on page 305

“syntaxonly” on page 327

## p

| Option Type  | Default Value | #pragma options |
|--------------|---------------|-----------------|
| <i>-flag</i> | -             | -               |

#### Syntax

-p

#### Purpose

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

If the **-qtbtable** option is not set, the **-p** option will generate full traceback tables.

#### Example

To compile `myprogram.c` so that it can be used with the AIX **prof** command, enter:

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

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

#### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Debugging Features” on page 227

“pg” on page 311 Compiler Option

**prof** command in the *AIX Version 4 Commands Reference*, for details on profiling.

## pascal

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <i>-qoption</i> | nopascal      | -               |

#### Syntax

-qpascal | -qnopascal



## Purpose

Ignores the word **pascal** in type specifiers and function declarations.

## Notes

This option can be used to improve compatibility of C for AIX programs on some other systems.

## Related Reading

“Compiler Options and Their Defaults” on page 218

## pdf1, pdf2

| Option Type           | Default Value    | #pragma options |
|-----------------------|------------------|-----------------|
| <code>-qoption</code> | nopdf1<br>nopdf2 | -               |

## Syntax

`-qpdf1` | `-qpdf2` | `-qnopdf1` | `-qnopdf2`

## Purpose

Tunes optimizations through Profile-Directed Feedback (PDF), where results from one or more sample program executions are used to improve optimization near conditional branches and in frequently executed code sections.

## Notes

To use PDF:

1. Compile some or all of the source files in a program with the **-qpdf1** option. **main** must be compiled. The “**I**” on page 286pdf option is required during the link step, the **-O3** option is recommended for optimization. Pay special attention to the compiler options used to compile the files, because you will need to use the same options later.
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. Recompile your program, using the same compiler options as before but changing **-qpdf1** to **-qpdf2**. Remember that **-L**, **-I**, and some others are linker options, and you can change them at this point. In particular, leave the **-lpdf** option out. 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 optimum performance, use the **-O3** option with all compilations when you use PDF (as in the example above). With **-O2** optimization, one of the most important PDF optimizations (moving code before branches to fill delay slots) is not done.

The profile is placed in the current working directory, or the directory named by the **PDFDIR** environment variable if that variable is set.

To avoid wasting compilation and execution time, make sure the **PDFDIR** environment variable is set to an absolute path; otherwise, you might run the application from the wrong directory so that it cannot locate the profile data files. If 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

- Do not mix PDF files created by the current version of C for AIX with PDF files created by previous versions.
- PDF optimizations also require at least level 2 of **-O**.
- The main program must be compiled with PDF for profiling to work properly. If you want to use this option to optimize a library or other code that does not usually incorporate a main program, supply a main program for the first PDF compilation, then omit the main program for the second PDF compilation.
- Do not compile or run two different applications that use the same **PDFDIR** directory at the same time.
- 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.
- 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 are available for managing the **PDFDIR** directory:

|                                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|----------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>resetpdf [pathname]</code> | Zeros out all profiling information (but does not remove the data files) from the pathname directory; or if pathname is not specified, from the <b>PDFDIR</b> directory; or if <b>PDFDIR</b> is not set, from the current directory.<br><br>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 <b>resetpdf</b> to reset the profiling information for the entire application, after making significant changes that may affect execution counts for parts of the program that were not recompiled. |
| <code>cleanpdf [pathname]</code> | Removes all profiling information from the pathname directory; or if pathname is not specified, from the <b>PDFDIR</b> directory; or if <b>PDFDIR</b> is not set, from the current directory.<br><br>Removing the profiling information reduces the runtime overhead if you change the program and then go through the PDF process again.<br><br>Run this program after compiling with <b>-qpdf2</b> , or after finishing with the PDF process for a particular application. If you continue using PDF with an application after running <b>cleanpdf</b> , you must recompile all the files with <b>-qpdf1</b> .                         |

### Example 1

Here are the steps for a simple example:

1. First, set the **PDFDIR** environment variable:

```
export PDFDIR=/home/user
```

2. Compile all files with **-qpdf1** and **-O3**, and link with **-lpdf**.

```
xlc -qpdf1 -lpdf -O3 file1.c file2.c file3.c -L/usr/vac/lib
```

3. Run with one set of input data:

```
a.out < sample.data
```

4. Recompile all files with **-qpdf2** and **-O3**:

```
xlc -qpdf2 -O3 file1.c file2.c file3.c
```

The program should now run faster than without PDF, if the sample used data was typical of actual program data.

**Note:** When using **-qpdf1**, specify the search location for its libraries with the **-L** compiler option, as shown in step 2 above.

### Example 2

Here are the steps for a more elaborate example.

1. Set the **PDFDIR** environment variable:

```
export PDFDIR=/home/user
```

2. Compile most of the files with **-qpdf1**.

```
xlc -qpdf1 -O3 -c file1.c file2.c file3.c -L/usr/vac/lib
```

3. This file is not so important to optimize:

```
xlc -c file4.c
```

4. Non-PDF object files like file4.o can be linked in:

```
xlc -qpdf1 -lpdf file1.o file2.o file3.o file4.o -L/usr/vac/lib
```

5. Run several times with different input data:

```
a.out < polar_orbit.data  
a.out < elliptical_orbit.data  
a.out < geosynchronous_orbit.data
```

6. You do not need to recompile the source of non-PDF object files:

```
xlc -qpdf2 -O3 file1.c file2.c file3.c
```

7. Link all the object files into the final application:

```
xlc file1.o file2.o file3.o file4.o
```

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

“L” on page 285

“l” on page 286

“O, optimize” on page 302

## pg

| Option Type  | Default Value | #pragma options |
|--------------|---------------|-----------------|
| <i>-flag</i> | -             | -               |

### Syntax

```
-pg
```

### 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.

### 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 Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Debugging Features” on page 227

“p” on page 308

**gprof** command in the *AIX Version 4 Commands Reference*, for details on profiling.

## phsinfo

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | nophsinfo     | -               |

### Syntax

```
-qphsinfo | -qnophsinfo
```

### Purpose

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

### Example

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

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

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Output” on page 228

## proclocal, procimported, procunknown

| Option Type     | Default Value | #pragma options                         |
|-----------------|---------------|-----------------------------------------|
| <b>-qoption</b> | proclocal*    | PROCLOCAL, PROCIMPORTED,<br>PROCUNKNOWN |

### Syntax

```
-qproclocal | -qproclocal=names
-qprocimported | -qprocimported=names
-qprocunknown | -qprocunknown=names
PROCLOCAL | PROCLOCAL=names
PROCIMPORTED | PROCIMPORTED=names
PROCUNKNOWN | PROCUNKNOWN=names
```

### Purpose

Marks functions as local, imported, or unknown.

### Default

The default is to assume that all functions whose definition is in the current compilation unit are local (**proclocal**), and that all other functions are unknown (**procunknown**). If any functions that are marked as local resolve to shared library functions, the linkage editor will detect the error and issue warnings 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. <b>-qprocllocal</b> changes the default to assume that all functions are local. <b>-qprocllocal=names</b> marks the named functions as local, where <i>names</i> 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. <b>-qprocimported</b> changes the default to assume that all functions are imported. <b>-qprocimported=names</b> marks the named functions as imported, where <i>names</i> is a list of function identifiers separated by colons (:). The default is not changed.                                                                                                                                                                                                                                                                                                             |
| Unknown functions  | 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. Are resolved to either statically or dynamically bound objects during link-editing. <b>-qprocunknown</b> changes the default to assume that all functions are unknown. <b>-qprocunknown=names</b> marks the named functions as unknown, where <i>names</i> 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 the functions **fun** and **sun** are specified as **local**, **moon** and **stars** are specified as **imported**, and **venus** is specified as **unknown**, enter:

```
xlc myprogram.c oldprogs.a -qprolocal=fun(int):sun()  
-qprocimported=moon():stars(float) -qprocunknown=venus()
```

## Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

## proto

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | noproto       | PROTO           |

## Syntax

```
-qproto | -qnoproto  
PROTO | NOPROTO
```

## Purpose

Assumes all functions are prototyped.

## 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 Reading

“Compiler Options and Their Defaults” on page 218

“info” on page 275 Compiler Option

## Q

| Option Type  | Default Value     | #pragma options |
|--------------|-------------------|-----------------|
| <i>-flag</i> | <i>See below.</i> | -               |

## Syntax

```
-Q | -Q=threshold | -Q=names | -Q=names | -Q!
```

## 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.

## 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 C for AIX **\_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, specify optimization (**-O**) and also specify the appropriate **-Q** option, as described below:

- 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=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 */
    }
}
```

`-Q-names` Does not inline functions listed by *names*. Separate each *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.

`-Q+names` A warning message is issued for functions that are not defined in the source file. Attempts to inline the functions listed by *names* and any other appropriate functions. Each *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+names` to inline specific functions. For example:

```
-Q=0
```

followed by:

```
-Q+salary:taxes:benefits
```

causes *only* the functions named **salary**, **taxes**, or **benefits** to be inlined, if possible, and no others.

## 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 specify the `-O` option (to optimize your program) and the `-Q` option (to inline functions), the compiler attempts to inline the functions you specify.

## Example

To compile the program `myprogram.c` so that no functions are inlined, enter:

```
xlc myprogram.c -O -Q!
```

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

```
xlc my_c_program.c -O -Q=12
```

### Related Reading

“Program Optimization with the C for AIX Compiler” on page 23

“Using Inlined Components” on page 202

“Writing Optimized Program Source Code” on page 197

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

“`_Inline`, `_inline`, `__inline`” on page 204, “`_Inline`, `_inline`, `__inline`” on page 204, and “`_Inline`, `_inline`, `__inline`” on page 204

“`g`” on page 267

“`O`, optimize” on page 302

“`inline`” on page 277

## r

| Option Type        | Default Value | #pragma options |
|--------------------|---------------|-----------------|
| <code>-flag</code> | -             | -               |

### Syntax

`-r`

### Purpose

Produces a relocatable object. This permits the output file to be produced even though it contains unresolved symbols.

### Notes

A file produced with this flag is expected to be used as a file parameter in another call to `xlc`.

### Example

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

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

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Linkage Options” on page 230

## rndsngl

| Option Type           | Default Value          | #pragma options      |
|-----------------------|------------------------|----------------------|
| <code>-qoption</code> | <code>norndsngl</code> | <code>RNDSNGL</code> |

### Syntax:

```
-qrndsngl | -qnorndsngl  
RNDSNGL | NORNDSNGL
```

### 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.

### 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. See “Floating-Point Compiler Options” on page 27 before you use the **-qrndsngl** option.

#### Related Reading

“Floating-Point Compiler Options” on page 27  
“Compiler Options and Their Defaults” on page 218  
“float” on page 261  
“hsflt” on page 271

## ro

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | ro*           | RO              |

#### Syntax:

```
-qro | -qnoro  
RO | NORO
```

#### Purpose

Specifies the storage type for string literals.

#### Default

The default with **xlc** and **c89** is **ro**. The default with **cc** is **nororo**.

#### Notes

If **ro** is specified, the compiler places string literals in read-only storage. If **nororo** is specified, string literals are placed in read/write storage.

You can also specify the storage type in your source program using:

```
#pragma strings storage_type
```

where *storage\_type* is **read-only** or **writable**.

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

#### Example

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

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

#### Related Reading

“Compiler Options and Their Defaults” on page 218  
“Options that Specify the Compiler Object Code Produced” on page 229

## roconst

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | roconst*      | ROCONST         |

#### Syntax

-qroconst | -qnorroconst  
ROCONST | NOROCONST

### Purpose

Specifies the storage location for constant values.

### Default

The default with **xlc** and **c89** is **roconst**. The default with **cc** is **norroconst**.

### Notes

If **-qroconst** is specified, the compiler places constants in read-only storage. If **-qnorroconst** 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 Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

“ro” on page 317

## rrm

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | norm          | RRM             |

### Syntax

-qrrm | -qnorrm  
RRM | NORRM

### Purpose

Prevents floating-point optimizations that are incompatible with run-time rounding to plus and minus infinity modes.

### 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, **-qnorm**, 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 Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

“y” on page 339

“float” on page 261

## S

| Option Type  | Default Value | #pragma options |
|--------------|---------------|-----------------|
| <i>-flag</i> | -             | -               |

### Syntax

`-S`

### Purpose

Generates an assembler language file (**.s**) for each source file. The resulting **.s** files can be assembled to produce object **.o** files or an executable file (**a.out**).

### Notes

You can invoke the assembler with the **xlc** command. For example,

```
xlc myprogram.s
```

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

If you specify **-S** with **-E** or **-P**, **-E** or **-P** takes precedence. Note the following order of precedence with respect to the **-S** option:

1. **-E** overrides **-P**
2. **-P** overrides **-S**
3. **-S** overrides **-c**

This order of precedence holds regardless of the order in which they were specified on the command line.

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

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

### Restrictions

The generated assembler files do not include all the data that is included in a **.o** file by the **-g** or **-qipa** options.

### Example

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

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

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

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

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

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

### Related Reading

“Compiler Options and Their Defaults” on page 218

“c” on page 242

“E” on page 253

“g” on page 267

“o” on page 305

“P” on page 307

“ipa” on page 279

*AIX Version 4 Assembler Language Reference*

*AIX Version 4 Files Reference*

## showinc

| Option Type           | Default Value | #pragma options |
|-----------------------|---------------|-----------------|
| <code>-qoption</code> | noshowinc     | SHOWinc         |

### Syntax

```
-qshowinc | -qnoshowinc  
SHOWINC
```

### Purpose

If used with **-qsource**, all the include files are included in the source listing.

### Example

To compile myprogram.c so that all included files appear in the source listing, enter:

```
xlc myprogram.c -qsource -qshowinc
```

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Output” on page 228

“source” on page 322

## smp

| Option Type           | Default Value | #pragma options |
|-----------------------|---------------|-----------------|
| <code>-qoption</code> | nosmp         | -               |

### Syntax

```
-qnosmp | -qsmp[=suboption[:suboption] [ ... ]]
```

### Purpose

Specifies if and how parallelized object code is generated, according to *suboption(s)* specified:

#### Suboption

#### Description

auto  
noauto

Enables or disables automatic parallelization.

**auto** is the default if **-qsmp** is specified without the **omp** suboption. Otherwise, the default is **noauto**.

explicit  
noexplicit

Enables or disables pragmas controlling explicit parallelization of countable loops.

**explicit** is the default.

If **noexplicit** is in effect, **#pragma ibm omp parallel\_loop** is not honored by the compiler.

nested\_par  
nonested\_par

Enables or disables parallelization of nested parallel constructs.

**nonested\_par** is the default. If one parallel construct is run as part of another parallel construct, the execution of the nested construct is serialized by the compiler for better performance.

If **nested\_par** is in effect, nested parallel constructs are not serialized.

**Notes:**

1. **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.
2. 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.

omp  
noomp

Enables or disables strict compliance with OpenMP C and C++ API specifications.

**noomp** is the default. This mode allows for maximum program parallelization, but may not be completely compliant to the OpenMP API specification.

If you specify the **omp** suboption, the compiler disables automatic parallelization and warns of directives that are not OpenMP-compliant. The `_OPENMP` macro is defined.

Certain other **smp** suboptions enable compiler parallelization features that do not comply with the OpenMP specification. If they are specified together with the **omp** suboption, a warning message issued. These suboptions are:

- auto
- nested\_par
- rec\_locks
- schedule=affinity=*n*

rec\_locks  
norec\_locks

Specifies whether recursive locks are used to implement critical sections.

If **rec\_locks** is in effect, recursive locks are used, and nested critical sections will not cause a deadlock.

The default is **norec\_locks**, or regular locks.

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.

Valid options for *sched\_type* are:

- dynamic[=*n*]
- guided[=*n*]
- static[=*n*]
- affinity[=*n*]
- runtime

If *sched\_type* is not specified, **runtime** is assumed as the default setting.

For more information about these scheduling algorithms, see `schedule` pragma.

## Notes

- Specifying **-qsmp** without suboptions is equivalent to specifying **-qsmp=auto:explicit:noomp:norec\_locks:nonested\_par:schedule=runtime**.
- 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** defines the `_IBMSMP` preprocessing macro
- Specifying **-qsmp** implicitly sets **-O2**. The **-qsmp** option overrides **-qnooptimize**, but does not override **-O3** or **-O4**.
- **-qsmp** must be used only with thread-safe compiler mode invocations such as `xlc_r`. These invocations ensure that the **pthread**s, **xlsmp**, and thread-safe versions of all default run-time libraries are linked to the resulting executable.

### Related Reading

- “Chapter 5. Program Parallelization” on page 37
- “Compiler Modes” on page 5
- “Using Pragmas to Control Parallel Processing” on page 41
- “Invoking the Compiler” on page 8
- “Compiler Options and Their Defaults” on page 218
- “Options that Specify the Compiler Object Code Produced” on page 229
- “#pragma Preprocessor Directives for Parallel Processing” on page 381
- “#pragma ibm schedule Preprocessor Directive” on page 386
- “Run-time Options for Parallel Processing” on page 402
- “Built-in Functions Used for Parallel Processing” on page 400
- “O, optimize” on page 302
- “ignprag” on page 274
- “O, optimize” on page 302
- “threaded” on page 331

## source

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | nosource      | SOURCE          |

### Syntax:

```
-qsource | -qnosource
SOURCE | NOSOURCE
```

### Purpose

Produces a compiler listing and includes source code.

### 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.

### Example

The following code causes the parts of the source code between the **#pragma options** directives to be included in the compiler listing:

```
#pragma options source
.
.
.
/* Source code to be included in the compiler listing
```

```

    is bracketed by #pragma options directives.
*/
    . . .
    #pragma options nosource

```

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

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

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Output” on page 228

“#pragma options Preprocessor Directive” on page 375

“noprnt” on page 301

## spill

| Option Type           | Default Value | #pragma options    |
|-----------------------|---------------|--------------------|
| <code>-qoption</code> | spill=512     | SPILL= <i>size</i> |

### Syntax

```

-qspill=size
SPILL=size

```

### Purpose

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

### Notes

If your program is very complex, or if there are too many computations to hold in registers at one time and your program needs temporary storage, you might need to increase this area. Do not enlarge the spill area unless the compiler issues a message requesting a larger spill area. In case of a conflict, the largest spill area specified is used.

### Example

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

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

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

## spnans

| Option Type           | Default Value | #pragma options |
|-----------------------|---------------|-----------------|
| <code>-qoption</code> | nospnans      | SPNANS          |

### Syntax

```

-qspnans | -qnospnans
SPNANS | NOSPNANS

```

### Purpose

Generates extra instructions to detect signalling NaN on conversion from single precision to double precision. The **nospnans** option specifies that this conversion need not be detected.

## Notes

The **-qhsflt** option overrides the **spnans** option

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

## Related Reading

“Compiler Options and Their Defaults” on page 218

“float” on page 261

“hsflt” on page 271

## srcmsg

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | nosrcmsg      | SRCMSG          |

## Syntax

```
-qsrcmsg | -qnosrcmsg  
SRCMSG | NOSRCMSG
```

## Purpose

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

## 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 (**nosrcmsg**) 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 Reading

“Compiler Message Format” on page 21

“Message Severity Levels and Compiler Response” on page 20

“Compiler Message and Listing Information” on page 18

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Output” on page 228

## statsym

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | nostatsym     | -               |

## Syntax

```
-qstatsym | -qnostatsym
```



## 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).

## 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 Reading

“Compiler Options and Their Defaults” on page 218

## stdinc

| Option Type           | Default Value | #pragma options |
|-----------------------|---------------|-----------------|
| <code>-qoption</code> | stdinc        | STDINC          |

## Syntax:

```
-qstdinc | -qnostdinc  
STDINC | NOSTDINC
```

## Purpose

Specifies which files are included with **#include** `<file_name>` and **#include** “`file_name`” directives.

## Notes

If you specify **-qnostdinc**, the compiler will not search the directory `/usr/include` (unless you explicitly add them with the **-Idirectory** option).

If a full (absolute) path name is specified, this option has no effect on that path name. It will still have an effect on all relative path names.

**-qnostdinc** is independent of **-qidirfirst**. (**-qidirfirst** searches the directory specified with **-Idirectory** before searching the directory where the current source file resides.

The search order for files is described in “Directory Search Sequence for Include Files Using Relative Path Names” on page 178.

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 Reading

“Compiler Options and Their Defaults” on page 218

“I” on page 272

“idirfirst” on page 273

## strict

| Option Type           | Default Value     | #pragma options |
|-----------------------|-------------------|-----------------|
| <code>-qoption</code> | <i>See below.</i> | STRICT          |

### Syntax

```
-qstrict | -qnostrict  
STRICT | NOSTRICT
```

### Purpose

Turns off aggressive optimizations that have the potential to alter the semantics of your program.

### Notes

`-qnostrict` has no effect at `-O` optimization level 0.

`-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.

Unless explicitly set otherwise by the `-qfloat` option:

- `-qstrict` sets `-qfloat=nofltint:norsqrt`.
- `-qnostrict` sets `-qfloat=fltint:rsqrt`.

You can use `-qfloat=fltint` and `-qfloat=rsqrt` to override the `-qstrict` settings.

For example:

- Using `-O3 -qstrict -qfloat=fltint` means that `-qfloat=fltint` 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`.

### Defaults

Default setting for the `strict` option varies according to `-O` optimization level in effect:

| Optimization level | Default setting for strict option |
|--------------------|-----------------------------------|
| 0                  | <code>-qstrict</code>             |
| 2                  | <code>-qstrict</code>             |
| 3                  | <code>-qnostrict</code>           |
| 4                  | <code>-qnostrict</code>           |

You can override the default settings by explicitly setting either `-qstrict` or `-qnostrict`. In the example below, `-qstrict` is active regardless of the `-O3` optimization level selected.

```
xlc myprogram.c -O3 -qstrict -qfloat=fltint:rsqrt
```

### Example

To compile `myprogram.c` so that the aggressive optimizations of `-O3` are turned off, range checking is turned off (`-qfloat=fltint`), 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=fltint:rsqrt
```

### Related Reading

“Compiler Options and Their Defaults” on page 218  
“#pragma option\_override Preprocessor Directive” on page 374  
“O, optimize” on page 302  
“float” on page 261  
“strict\_induction”

## strict\_induction

| Option Type           | Default Value     | #pragma options | C | C++ |
|-----------------------|-------------------|-----------------|---|-----|
| <code>-qoption</code> | <i>See below.</i> | -               | x | x   |

### Syntax

`-qstrict_induction` | `-qnostrict_induction`

### Purpose

Setting **-qstrict\_induction** 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.

### Notes

This option affects only loops which have an induction (loop counter) variable declared as a different size than a register. The most probable incidence of such a situation will likely involve using 32-bit loop counters (int or unsigned int) when compiling in 64-bit mode. Unless you intend such variables to overflow or wrap around, use **-qnostrict\_induction**.

Using **-qstrict\_induction** can cause considerable performance degradation. However, the option may be useful for debugging a program sensitive to variable overflow or wrap-around.

### Default

- **-qstrict\_induction** with optimization level 0, or when using **c89** compiler invocation mode.
- **-qnostrict\_induction** otherwise.

### Related Reading

“Compiler Modes” on page 5  
“Compiler Options and Their Defaults” on page 218

## syntaxonly

| Option Type           | Default Value | #pragma options |
|-----------------------|---------------|-----------------|
| <code>-qoption</code> | -             | -               |

### Syntax

`-qSYNTAXonly`

### Purpose

Causes the compiler to perform syntax checking without generating an object file.

### 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, precompiled header files, etc) are still produced if their corresponding options are set.

## Example

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**” on page 305 option so no object file is produced.

## Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Debugging Features” on page 227

“C” on page 242

“E” on page 253

“P” on page 307

## suppress

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | nosuppress    | -               |

## Syntax

```
-qsuppress=msg_num[:msg_num ...] | -qnosuppress
```

## Purpose

This compiler option lets you specify warning or information messages to be suppressed in compiler listings or screen displays.

## Notes

This option suppresses compiler messages only, and has no effect on linker or operating system messages.

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

Assuming a sample program called myprogram.c, shown below:

```
#pragma incorrect_pragma
void () {
}
```

Compiling the program above would normally result in the following or similar compiler message:

```
"t.c", line 1.1: 1506-224 (I) Incorrect #pragma ignored
```

To suppress this message, compile the sample program with the **-qsuppress** option as follows:

```
xlc myprogram.c -qsuppress=1506-224
```

### Related Reading

“Compiler Options and Their Defaults” on page 218  
“flag” on page 261  
“halt” on page 269  
“maxerr” on page 299

## t

| Option Type  | Default Value     | #pragma options |
|--------------|-------------------|-----------------|
| <i>-flag</i> | <i>See below.</i> | -               |

### Syntax:

*-tprograms*

### Purpose

Adds the prefix specified by the **-B** option to the designated *programs*.

### Notes

This option can only be used with the **-B** option. The flags representing the standard *program* names are:

| Programs | 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                                |
| m        | Linkage helper ( <b>munch</b> )               |

### 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 Reading

“Compiler Options and Their Defaults” on page 218  
“B” on page 239

## tabsize

| Option Type     | Default Value          | #pragma options |
|-----------------|------------------------|-----------------|
| <i>-qoption</i> | <code>tabsize=8</code> | -               |

### Syntax

-qtabsize=*n*

### Purpose

Changes the length of tabs as perceived by the compiler.

### Notes

*n* is the number of character spaces representing a tab in your source program.

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 Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Characteristics” on page 226

## tbtable

| Option Type | Default Value | #pragma options |
|-------------|---------------|-----------------|
| -qoption    | full*         | TBTABLE         |

### Syntax

-qtbttable=*suboption*  
TBTABLE=*suboption*

### 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.

### Notes

Values for *suboption* are:

|       |                                                                                                                  |
|-------|------------------------------------------------------------------------------------------------------------------|
| none  | No traceback table is generated. The stack frame cannot be unwound.                                              |
| full  | A full traceback table is generated, complete with name and parameter information.                               |
| small | The traceback table generated has no name or parameter information, but otherwise has full traceback capability. |

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

### Default

Many performance measurement tools require a full traceback table to properly analyze optimized code. The **/etc/vac.cfg** 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 **/etc/vac.cfg** compiler configuration file:

1. Remove the **-qtable=full** option from the **options** lines of the C compilation stanzas.
2. Remove the **-qtable=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, **-qtbttable=small**
- When compiling with no optimization options set, **-qtable=full**

See “Interlanguage Calls - Traceback Table” on page 214 for a brief description of traceback tables. The AIX Version 4 traceback mechanism is described in the “Subroutine Linkage Convention” section of the

**Related Reading**

“Compiler Options and Their Defaults” on page 218  
 “Options that Specify Debugging Features” on page 227  
 “etc/vac.cfg - Default Configuration File” on page 474

**threaded**

| Option Type     | Default Value     | #pragma options |
|-----------------|-------------------|-----------------|
| <b>-qoption</b> | <i>See below.</i> | -               |

**Syntax**

`-qthreaded` | `-qnothreaded`

**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.

**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.

**Default**

The default is **-qthreaded** when compiling with `_r` invocation modes, and **-qnothreaded** when compiling with other invocation modes.

**Related Reading**

“Compiler Modes” on page 5  
 “Compiler Options and Their Defaults” on page 218  
 “Options that Specify the Compiler Object Code Produced” on page 229  
 “smp” on page 320

**tune**

| Option Type     | Default Value     | #pragma options        |
|-----------------|-------------------|------------------------|
| <b>-qoption</b> | <i>See below.</i> | TUNE= <i>suboption</i> |

**Syntax**

`-qtune=suboption`  
`TUNE=suboption`

**Purpose**

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

**Notes**

Allowable values for *suboption* are:

`auto` Automatically detects the specific architecture of the compiling machine. Use this suboption only if the execution environment is the same as the compilation environment.

|       |                                                                                                                                                          |
|-------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| 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 POWER3 processors.                                                                                                    |
| pwrx  | Produces object code optimized for the POWER2 hardware platforms (same as <b>-qtune=pwr2</b> ).                                                          |
| rs64a | Produces object code optimized for the RS64A processor.                                                                                                  |
| rs64b | Produces object code optimized for the RS64B processor.                                                                                                  |

If **-qtune** is specified without **-qarch=suboption**, the compiler uses **-qarch=com**.

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.

### 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 “**Acceptable Compiler Mode and Processor Architecture Combinations**” on page 16.

### 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 Reading

- “Specifying Compiler Options for Architecture-Specific, 32- or 64-bit Compilation” on page 14
- “Acceptable Compiler Mode and Processor Architecture Combinations” on page 16
- “Compiler Options and Their Defaults” on page 218
- “Options that Specify the Compiler Object Code Produced” on page 229
- “arch” on page 237

## U

| Option Type  | Default Value | #pragma options |
|--------------|---------------|-----------------|
| <i>-flag</i> | -             | -               |

### Syntax

```
-Uname
```



## Purpose

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

## Notes

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

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

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

## Example

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

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

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

## Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Preprocessor Options” on page 228

“D” on page 250

## unroll

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | unroll=4*     | -               |

## Syntax

```
-qunroll=n | -qnounroll
```

## Purpose

Unrolls inner loops in the program by a factor of *n*.

## Notes

When **-qunroll** is specified, the bodies of inner loops will be duplicated *n*-1 times, creating a loop with *n* original bodies. The loop control may be modified in some cases to avoid unnecessary branching.

The maximum value for *n* is 8.

## Default

The compiler will perform automatic unrolling of inner loops by a factor of 4 at an optimization level of 2 or higher (for example, when the **-O3** optimizing option is specified). This will be disabled, however, if **-qnounroll** is specified at the same time.

## Example

In the following example, loop control is not modified:

```
while (*s != 0)
{
    *p++ = *s++;
}
```

Unrolling this by a factor of 2 gives:

```

while (*s)
{
    *p++ = *s++;
    if (*s == 0) break;
    *p++ = *s++;
}

```

In this example, loop control *is* modified:

```

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 Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

“O, optimize” on page 302

## upconv

| Option Type | Default Value | #pragma options |
|-------------|---------------|-----------------|
| -qoption    | noupconv*     | UPCONV          |

### Syntax

```

-qupconv | -qnoupconv
UPCONV | NOUPCONV

```

### Purpose

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

### 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 an **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 Reading

“Compiler Options and Their Defaults” on page 218

“langlv” on page 286

## usepcomp

| Option Type     | Default Value | #pragma options |
|-----------------|---------------|-----------------|
| <b>-qoption</b> | nousepcomp    | -               |

### Syntax

```
-qusepcomp | -qusepcomp=directory | -qnousepcomp
```

### Purpose

Uses a precompiled header file if no included files that have not changed since the precompiled header was created. This may help improve compile time.

### Notes

Usage modes for **usepcomp** are:

|                                   |                                                                                                                                                                                                                                                                                                                                                                                    |
|-----------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-qusepcomp</code>           | Uses the precompiled header file called <b>csetc.pch</b> , if it exists in the current directory.                                                                                                                                                                                                                                                                                  |
| <code>-qusepcomp=directory</code> | Uses a precompiled header file if: <ul style="list-style-type: none"><li>• <i>directory</i> is the name of an existing directory, and the <b>csetc.pch</b> precompiled header file exists in that directory.</li><li>• a directory with the name <i>directory</i> does not exist, but a precompiled header file called <i>directory</i> exists in the current directory.</li></ul> |
| <code>-qnousepcomp</code>         | Does not use precompiled header files.                                                                                                                                                                                                                                                                                                                                             |

The **-qusepcomp** and **-qgenpcomp** options are designed to be used together, but they may be used separately.

- **-qgenpcomp** used alone will refresh the contents of the precompiled header file, even if it already exists. This is useful if the file has been corrupted.
- **-qusepcomp** used alone will use an existing precompiled header file without creating a new one. This is useful if you only want do not want the precompiled header file to be recompiled, or if remaining disk space is low.

When **-qusepcomp** and **-qgenpcomp** are used together, the compiler will automatically maintain and use a current precompiled header.

If you update your system header files, you can regenerate them with the `/usr/vac/bin/mkpcmp` command.

Precompiled headers will only be used at the same language level used during their creation.

For a given `#include`, `-qusepcmp` is checked first. Then the compiler checks for a precompiled version of the file to be included if such is specified. If it is found and it is current, it is used.

If a precompiled header is not being used (for example, if a current one is not found, or if `-qusepcmp` is not specified), and `-qgenpcmp` is specified, the compiler will create a new precompiled header (even if it exists and is current).

The precompiled headers created by installing C for AIX are listed in the LPP inventory, and are removed if you uninstall C for AIX. Any additional headers you create are *not* removed during uninstall.

### Related Reading

“Creating and Using Precompiled Headers” on page 35

“Compiler Options and Their Defaults” on page 218

“#include Preprocessor Directive” on page 356

“genpcmp” on page 267

## V

| Option Type  | Default Value | #pragma options |
|--------------|---------------|-----------------|
| <i>-flag</i> | -             | -               |

### Syntax

`-v`

### Purpose

Instructs the compiler to report information on the progress of the compilation, and names the programs being invoked within the compiler and the options being specified to each program. Information is displayed to standard output.

### 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 Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Output” on page 228

“#” on page 231

## W

| Option Type  | Default Value | #pragma options |
|--------------|---------------|-----------------|
| <i>-flag</i> | -             | -               |

### Syntax

*-Wprogram, options*

### Purpose

Passes the listed options to the designated compiler *program*.

*program* can be:

| 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:

```
xlc myprogram.c -Wl:a, -pg
```

In a configuration file, use the `\,` sequence to represent the comma (`,`).

```
-Wl:a\,-pg
```

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Characteristics” on page 226

## W

| Option Type  | Default Value     | #pragma options |
|--------------|-------------------|-----------------|
| <i>-flag</i> | <i>See below.</i> | -               |

### Syntax

`-w`

### Purpose

Requests that warnings and lower-level messages be suppressed. Specifying this option is equivalent to specifying **-qflag=e:e**.

### Example

To compile `myprogram.c` so that no warning messages are displayed, enter:

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

### Related Reading

“Compiler Options and Their Defaults” on page 218

“Options that Specify Compiler Output” on page 228

“flag” on page 261

## warn64

| Option Type           | Default Value | #pragma options |
|-----------------------|---------------|-----------------|
| <code>-qoption</code> | nowarn64      | -               |

### Syntax

`-qwarn64`

### Purpose

Enables checking for possible *long-to-integer* truncation.

### 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. The 64-bit compiler mode , such as possible:

- 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 Reading

“Specifying Compiler Options for Architecture-Specific, 32- or 64-bit Compilation” on page 14

“Acceptable Compiler Mode and Processor Architecture Combinations” on page 16

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229

## xcall

| Option Type           | Default Value | #pragma options |
|-----------------------|---------------|-----------------|
| <code>-qoption</code> | noxcall       | -               |

### Syntax

`-qxcall | -qnoxcall`

### Purpose

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

### 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 Reading

"Compiler Options and Their Defaults" on page 218

## xref

| Option Type | Default Value | #pragma options |
|-------------|---------------|-----------------|
| -qoption    | noxref        | XREF            |

### Syntax

-qxref | -qnoxref  
XREF | NOXREF

### Purpose

Produces a compiler listing that includes a cross-reference listing of all identifiers.

### Notes

Usage modes for **xref** are:

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

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

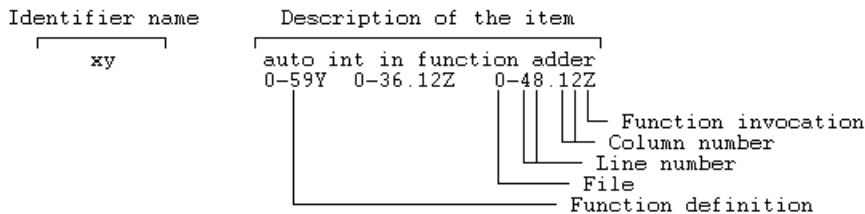
Any function defined with the **#pragma mc\_func** *function\_name* directive is listed as being defined on the line of the **#pragma** directive.

### 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
```

A typical cross-reference listing has the form:



### Related Reading

"Compiler Options and Their Defaults" on page 218  
"Options that Specify Compiler Output" on page 228  
"noprint" on page 301

## y

| Option Type | Default Value | #pragma options        |
|-------------|---------------|------------------------|
| -flag       | -             | <i>Y rounding_mode</i> |

### Syntax

`-rounding_mode` *Yrounding\_mode*

### **Purpose**

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

### **Notes**

*rounding\_mode* must be one of the following:

|          |                                                                 |
|----------|-----------------------------------------------------------------|
| <b>n</b> | Round to the nearest representable number. This is the default. |
| <b>m</b> | Round toward minus infinity.                                    |
| <b>p</b> | Round toward plus infinity.                                     |
| <b>z</b> | Round toward zero.                                              |

### **Example**

To compile `myprogram.c` so that constant floating-point expressions are rounded toward zero at compile time, enter:

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

### **Related Reading**

“Compiler Options and Their Defaults” on page 218

“Options that Specify the Compiler Object Code Produced” on page 229



## Appendix B. 32-bit to 64-bit Migration Considerations

This section outlines various portability considerations in moving C programs from 32-bit to 64-bit mode.

- Constants (page 341)
- Undeclared Functions (page 342)
- Assignment of Long Types to Integer and Pointers (page 342)
- Structure Sizes and Alignment (page 343)
- Bitfields (page 343)
- Miscellaneous (page 343)
- Interlanguage Calls with Fortran (page 344)

### Constants

The limits of constants change. This table shows changed items in the **limits.h** header file, their hexadecimal value, and decimal equivalent. The equation gives an idea of how to construct these values.

| Type                             | Hexadecimal          | Equation                    | Decimal                     |
|----------------------------------|----------------------|-----------------------------|-----------------------------|
| signed long min<br>(LONG_MIN)    | 0x8000000000000000L  | $-(2^{63})$                 | -9,223,372,036,854,775,808  |
| signed long max<br>(LONG_MAX)    | 0x7FFFFFFFFFFFFFFFL  | $2^{63}-1$<br>(-LONG_MIN-1) | +9,223,372,036,854,775,807  |
| unsigned long max<br>(ULONG_MAX) | 0xFFFFFFFFFFFFFFFFUL | $2^{64}-1$                  | +18,446,744,073,709,551,616 |

In C, type identification of constants follows explicit rules. However, programs that use constants exceeding the limit (relying on a 2's complement representation) will experience unexpected results in the 64-bit mode. This is especially true of hexadecimal constants and unsuffixed constants, which are more likely to be extended into the 64-bit long type.

Problematic behaviors will generally occur at boundary areas such as:

- constant  $\geq$  UINT\_MAX
- constant  $<$  INT\_MIN
- constant  $>$  INT\_MAX

Some examples of undesirable boundary side effects are:

| Constant assigned to long      | 32 bit mode    | 64 bit mode    |
|--------------------------------|----------------|----------------|
| -2,147,483,649 (INT_MIN-1)     | +2,147,483,647 | -2,147,483,649 |
| +2,147,483,648 (INT_MAX+1)     | -2,147,483,648 | +2,147,483,648 |
| +4,294,496,726 (UINT_MAX+1)    | 0              | +4,294,967,296 |
| 0xFFFFFFFF (UINT_MAX)          | -1             | +4,294,496,295 |
| 0x100000000 (UINT_MAX+1)       | 0              | +4,294,967,296 |
| 0xFFFFFFFFFFFFFFFF (ULONG_MAX) | -1             | -1             |

Currently, the compiler gives out of range warning messages when attempting to assign a value larger than the designated range into a long type. The warning message is:

```
1506-207 (W) Integer constant 0x100000000 out of range.
```

This warning message may not appear for every case.

When you bit left-shift a 32-bit constant and assign it into a long type, signed values are sign-extended and unsigned values are zero-extended. The examples in the table below show the effects of performing a bit-shift on both 32- and 64-bit constants, using the following code segment:

```
long l=constantL<<1;
```

| Initial Constant Value | Constant Value after Bit-Shift |              |
|------------------------|--------------------------------|--------------|
|                        | 32-bit                         | 64-bit       |
| 0x7FFFFFFFL (INT_MAX)  | 0xFFFFFFFFE                    | 0xFFFFFFFFE  |
| 0x80000000L (INT_MIN)  | 0                              | 0x100000000  |
| 0xFFFFFFFFL (UINT_MAX) | 0xFFFFFFFFE                    | 0x1FFFFFFFFE |

Unsuffix constants can lead to type ambiguity that can impact other parts of your program, such as the result of **sizeof** operations. For example, in 32-bit mode the compiler types a number like 4294967295 (UINT\_MAX) as an unsigned long. In 64-bit mode, this same number becomes a signed long. To avoid this possibility, explicitly add a suffix to all constants that have the potential of impacting constant assignment or expression evaluation in other parts of your program. The fix for the above case is to write the number as 4294967295U. This forces the compiler to always see that constant as an unsigned int regardless of compiler mode.

### Assignment of Long Variables to Integers and Pointers

Using int and long types in expressions and assignments can lead to implicit conversion through promotions and demotions, or explicit conversions through assignments and argument passing. The following should be avoided:

- Using integer and long types interchangeably, leading to truncation of significant digits or unexpected results.
- Passing long arguments to functions expecting type int
- Exchanging pointers and int types, causing segmentation faults.
- Passing pointers to a function expecting an int type, resulting in truncation.
- Assignment of long types to float, causing possible loss of accuracy.

Assigning a long constant to an integer will cause truncation without warning. For example:

```
int i;  
long l=2147483648; /* INT_MAX+1*/  
i=l;
```

What will be the value of i? INT\_MAX+1 is 2147483647+1 (0x80000000), which becomes INT\_MIN when assigned into a signed type. Truncation occurs because the highest bit is treated as a sign bit. The rule here is that there will be a loss of significant digits.

Similar problems occur when passing constants directly to functions, and in functions that return long types. Making explicit use of the L and UL suffix will avoid most, but not all, problems. Alternately, you can avoid accidental conversions by using explicit prototyping. Another good practice is to avoid implicit type conversion by using explicit type casting to change types.

## Undeclared Functions

Any function that returns a pointer should be explicitly declared when compiling in 64-bit mode. Otherwise, the compiler will assume the function returns an int and truncate the resulting pointer, even if you were to assign it into a valid pointer.

Code such as:

```
a=(char *) calloc(25);
```

which used to work in 32-bit mode will in 64-bit mode will now silently get a truncated pointer. Even the type casting will not avoid this because the `calloc` has already been truncated after the return.

The fix in this case is to include the appropriate header file, which is `stdlib.h` and not `malloc.h`.

## Structure Sizes and Alignments

Structures may face potential porting problems.

The 64-bit specification changes the size, member and structure alignment of all structures that are recompiled in 64-bit mode. Structures with long types and pointers will generally change size and alignment in 64-bit mode. Some structures may not change in size because they happen to fall on an exact 8-byte boundary even in 32-bit mode.

Sharing data structures between 32- and 64-bit processes is no longer possible unless the structure is devoid of pointer and long types. Unions that attempt to share long and int types, or overlay pointers onto int types will now be aligned differently, or be corrupted. In general, all but the simplest structures must be checked for alignment and size dependencies.

The alignment for `-qalign=full`, `power` or `natural` changes for 64-bit mode. Structure members are aligned on their natural boundaries. Long types and pointer types are word-aligned in 32-bit mode, and doubleword aligned in 64-bit mode. Additional spaces could be used for padding members.

The alignment for `-qalign=twobyte` and `-qalign=mac68k` are not supported in 64-bit mode.

Structures are aligned according to the strictest aligned member. This remains unchanged from 32-bit mode. Because of the padding introduced by the member alignment, structure alignment may not be exactly the same as in the 32-bit mode. This is especially important when you have arrays of structures which contain pointer or long types. The member alignment will change, most likely leading to the structure alignment to change to doubleword alignment (if there are no long long types, double types and long double types).

## Bitfields

Structure bitfields are limited to 32 bits, and can be of type signed int, unsigned int or plain int. Bit fields are packed into the current word. Adjacent bit fields that cross a word boundary will start at storage unit. This storage unit is a word in `power` and `full` alignment, halfword in the `mac68k` and `twobyte` alignment, and byte in the `packed` alignment. 64-bit bitfields are not supported.

In 32-bit mode, non-integer bitfields are tolerated (but not respected) only in the C extended language level.

If you use long bit fields in 64-bit mode, their exact alignment may change in future versions of the compiler, even if the bitfield is under 32 bits in length.

## Miscellaneous Issues

- The **sizeof** operator will now return **size\_t** which is an unsigned long.
- The length of the integer required to hold the difference between two pointers is **ptrdiff\_t**, and is a signed long type.
- Masks will generally lead to different results when compiled in 64-bit mode from their 32-bit mode behavior.
- Many include files have pointers and structures in them, and their inclusion in 64-bit mode will change the size of your data section even if your program does not use structures and pointers explicitly.
- **\_\_int64** is a long type in 64-bit mode, but will look like a long long type in 32-bit mode. **\_\_int64** types can participate in promotion rules and arithmetic conversion when in 64-bit mode. When in 32-bit mode, these types can not participate in the usual arithmetic conversions.
- In 64-bit mode, member values in a structure passed by value to a **va\_arg** argument may not be accessed properly if the size of the structure is not a multiple of 8-bytes. This is a known limitation of the operating system.
- In 64-bit extended mode, zero-extension from unsigned int to an unsigned long preserves the bit pattern. For example, zero-extending an unsigned int with value 0xFFFF FFFF (large negative value) results in an unsigned long with value 0x0000 0000 FFFF FFFF (large positive value).

**Interlanguage Calls with Fortran**A significant number of applications use C, C++, and Fortran together, by calling each other or sharing files. Such applications are among the early candidates for porting to 64-bit platforms for its abilities to solve larger mathematical models. Experience shows that it is easier to modify data sizes/types on the C side than the Fortran side of such applications. The following table lists the equivalent Fortran type in the different modes.

| C/C++ type    | 32-bit  | 64-bit    |
|---------------|---------|-----------|
| int           | INTEGER | INTEGER   |
| unsigned int  | LOGICAL | LOGICAL   |
| signed long   | INTEGER | INTEGER*8 |
| unsigned long | LOGICAL | LOGICAL*8 |
| pointer       | INTEGER | INTEGER*8 |

A user must not mix XCOFF object formats from different modes. A 32-bit Fortran XCOFF cannot mix with a 64-bit C or C++ XCOFF object and vice versa. Since Fortran77 usually does not have an explicit pointer type, it is common practice to use INTEGER variables to hold C or C++ pointers in 32-bit mode. In 64-bit mode, the user should use INTEGER\*8 in Fortran. Fortran90 does have a pointer, but it is unsuitable for conversion to the basic C and C++ types.

In 64-bit mode, Fortran will have a POINTER\*8 that is 8 bytes in length as compared to their POINTER which is 4-bytes in length.

## Related Reading

“Appendix C. Operating System Migration Considerations” on page 345

---

## Appendix C. Operating System Migration Considerations

You should be aware of the following considerations when moving programs to AIX 4.3:

- **time\_t** has changed type from AIX 4.2 to AIX 4.3  
Library functions which take an argument of **time\_t** or return type **time\_t** may find type mismatches with your existing code in 32-bit mode. **time\_t** is changed from long type in AIX 4.2 to int type in AIX 4.3. The change in types may cause compile-time errors in your programs.
- **MB\_CUR\_MAX** has changed from **int** to **size\_t** in AIX 4.3  
**MB\_CUR\_MAX** is a macro defined in `stdlib.h` that calls `_getmbcurmax( )`. This function now returns **size\_t** which has always been **unsigned long**. In AIX 4.2, it was prototyped to return an **int**.
- **setlocale** in 64-bit mode  
If you have user locales defined, you must recompile them in 64-bit mode using **localedef**. This generates 32-bit and 64-bit versions of your locale file. Otherwise, calling **setlocale** in 64-bit mode will not find the user-defined locale file.  
However, **localedef** in AIX 4.3 supports only the charmap that is supplied with the AIX 4.3 distribution. If you need the charmaps from an older AIX distribution, you must explicitly copy them into your directory before using **localedef** with your custom locale definition file.  
In addition, **localedef** by default is set up to use `/bin/cc` and `/usr/bin/cc`. The C for AIX compiler does not create links in `/usr/bin` or `/bin` by default. Since **localedef** requires the use of a 64-bit compiler, you need to run `/usr/vac/bin/replaceCSet` to create links pointing to the C for AIX product. Invoke **localedef**, then execute `restoreCSet` to restore the links as they were before.
- The **make** tool does not discriminate between object formats  
The **make** tool only discriminates on the timestamp of files. The only case where this can cause problems is when you try to add same-named 32 and 64-bit objects into the archive. Running **make** first in 32-bit mode, then in 64-bit mode, will not update the 2nd object. Make only checks the timestamp of the first object it finds with the correct name.
- **int64** is type defined in `inttypes.h`  
If you use **int64** as a variable name, this is now a typedef in `inttypes.h`
- Header file predefined types that are based on **long**  
There are many header file predefined types, such as **size\_t** and **ptrdiff\_t**, which remain the same type regardless of 32 or 64-bit compiler mode. This presents a subtle opportunity for differences when compiling the same code in different mode of the compiler.  
Although **size\_t** remains the same type (**unsigned long**), the length of **size\_t** will change between different modes of AIX. This can cause library functions that return or take **size\_t** to change behavior in 32-bit to 64-bit mode. Specifically, **sizeof** will return an 8-byte value in 64-bit and a 4-byte value in 32-bit mode. The same applies to **ptrdiff\_t**, which is **signed long** in both modes.
- **m:n** thread may exhaust memory rapidly  
The **m:n** thread model is one of the 3 models used to map user threads to kernel threads.
  - In the **m:1** model, all user threads are mapped to one kernel thread, and all user threads run on one virtual processor. This is the traditional model on single-threaded systems.
  - In the **1:1** model, each user thread is mapped to one kernel thread, and each user thread runs on one virtual processor. POSIX 1003.1c Draft 7-based applications continue to run in **1:1** mode.
  - In the **m:n** model, all user threads are mapped to a pool of kernel threads, and all user threads run on a pool of virtual processors. One user thread may be bound to a specific virtual processor (like **1:1**) with remaining threads using the remaining virtual processors in the pool. This is the newest and most complex model. It is the default for XPG-5.

Previously, AIX 4.3.0 XPG-5 based applications ran in **1:1** mode. The same application now runs in **m:n** mode in AIX 4.3.1. The application should continue to function correctly, however, the performance of the application is likely to change.

The AIXTHREAD\_SCOPE environment variable resets the disposition of the default attribute. This can be used to change the scheduling policy from **m:n** to **1:1** or vice versa. Settings for this environment variable are:

```
AIXTHREAD_SCOPE=sched_policy
```

where *sched\_policy* is one of:

- |   |                                           |
|---|-------------------------------------------|
| P | - process based scheduling ( <b>m:n</b> ) |
| S | - system based scheduling ( <b>1:1</b> )  |

The AIXTHREAD\_SCOPE environment variable can also be used to overcome problems associated with the **m:n** based scheduling. APAR IX76628 is available to fix these problems.

We recommend setting the environment variable to S if your threaded application encounters problems.

#### **Related Reading**

“Appendix B. 32-bit to 64-bit Migration Considerations” on page 341

---

## Appendix D. Preprocessor Directives and Related Information

---

### List of Standard Preprocessor Directives

This page lists and briefly describes preprocessor directives available to you with C for AIX. To get more information on any item listed here, go to the reference page for that item.

| Preprocessor Directives                          |                                                                                                                              |
|--------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| Name                                             | Action                                                                                                                       |
| "# (Null) Preprocessor Directive"                | Null directive specifying that no action be performed.                                                                       |
| "#define Preprocessor Directive" on page 348     | Defines a preprocessor macro.                                                                                                |
| "#if, #elif Preprocessor Directives" on page 352 | Conditionally includes source text if the previous <b>#if</b> , <b>#ifdef</b> , <b>#ifndef</b> , or <b>#elif</b> test fails. |
| "#else Preprocessor Directive" on page 353       | Conditionally includes source text if the previous <b>#if</b> , <b>#ifdef</b> , <b>#ifndef</b> , or <b>#elif</b> test fails. |
| "#endif Preprocessor Directive" on page 353      | Ends conditional text.                                                                                                       |
| "#error Preprocessor Directive" on page 354      | Defines text for a compile-time error message.                                                                               |
| "#if, #elif Preprocessor Directives" on page 352 | Conditionally includes or suppresses portions of source code, depending on the result of a constant expression.              |
| "#ifdef Preprocessor Directive" on page 354      | Conditionally includes source text if a macro name is defined.                                                               |
| "#ifndef Preprocessor Directive" on page 355     | Conditionally includes source text if a macro name is not defined.                                                           |
| "#include Preprocessor Directive" on page 356    | Inserts text from another source file.                                                                                       |
| "#line Preprocessor Directive" on page 357       | Supplies a line number for compiler messages.                                                                                |
| "#pragma Preprocessor Directives" on page 363    | Specifies implementation-defined instructions to the compiler.                                                               |
| "#undef Preprocessor Directive" on page 358      | Removes a preprocessor macro definition.                                                                                     |

#### Related Reading

- "Preprocessor Directives" on page 58
- "Preprocessing Operations" on page 59
- "Preprocessor Macros" on page 59
- "Conditional Compilation Directives" on page 60
- "#pragma Preprocessor Directives" on page 363
- "#pragma Preprocessor Directives for Parallel Processing" on page 381

### # (Null) Preprocessor Directive

The *null directive* performs no action. It consists of a single # on a line of its own.

The null directive should not be confused with the # operator or the character that starts a preprocessor directive.

In the following example, if MINVAL is a defined macro name, no action is performed. If MINVAL is not a defined identifier, it is defined 1.

```
#ifdef MINVAL
#
#else
#define MINVAL 1
#endif
```

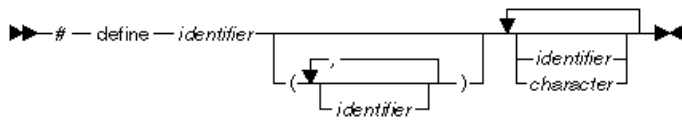
### Related Reading

"Preprocessor Directives" on page 58

"List of Standard Preprocessor Directives" on page 347

## #define Preprocessor Directive

A *preprocessor define directive* directs the preprocessor to replace all subsequent occurrences of a macro with specified replacement tokens.





The **#define** directive can contain an object-like definition or a function-like definition

### Object-Like Macros

An *object-like macro definition* replaces a single identifier with the specified replacement tokens. The following object-like definition causes the preprocessor to replace all subsequent instances of the identifier `COUNT` with the constant `1000`:

```
#define COUNT 1000
```

If the statement

```
int array[COUNT];
```

appears after this definition and in the same file as the definition, the preprocessor would change the statement to

```
int array[1000];
```

in the output of the preprocessor.

Other definitions can make reference to the identifier `COUNT`:

```
#define MAX_COUNT COUNT + 100
```

The preprocessor replaces each subsequent occurrence of `MAX_COUNT` with `COUNT + 100`, which the preprocessor then replaces with `1000 + 100`.

If a number that is partially built by a macro expansion is produced, the preprocessor does not consider the result to be a single value. For example, the following will not result in the value `10.2` but in a syntax error.

```
#define a 10  
a.2
```

Using the following also results in a syntax error:

```
#define a 10  
#define b a.11
```

To have the preprocessor treat the result as a single value, preprocess your source files using the **-P** compiler option and then compile the resulting `.i` file.

Identifiers that are partially built from a macro expansion may not be produced. Therefore, the following example contains two identifiers and results in a syntax error:

```
#define d efg  
abcd
```

## Function-Like Macros

To define a *function-like macro*, specify an identifier name followed by a parenthesized parameter list in parenthesis and the replacement tokens. The parameters are imbedded in the replacement code. White space cannot separate the identifier (which is the name of the macro) and the left parenthesis of the parameter list. A comma must separate each parameter. For portability, you should not have more than 31 parameters for a macro.

Use function-like macros in your program as follows. In the body of your program source, insert a defined function-like macro name followed by a list of arguments in parentheses. A comma must separate each argument. Once the preprocessor identifies a function-like macro invocation, argument substitution takes place. Parameters in the replacement code are replaced by the corresponding arguments. Any macro invocations contained in an argument itself are completely replaced before the argument replaces its corresponding parameter in the replacement code.

### Examples of Usage

The following line defines the macro SUM as having two parameters a and b and the replacement tokens (a + b):

```
#define SUM(a,b) (a + b)
```

This definition causes the preprocessor to change the following statements (if the statements appear after the previous definition):

```
c = SUM(x,y);  
c = d * SUM(x,y);
```

In the output of the preprocessor, these statements would appear as:

```
c = (x + y);  
c = d * (x + y);
```

Use parentheses to ensure correct evaluation of replacement text. For example, the definition:

```
#define SQR(c) ((c) * (c))
```

requires parentheses around each parameter c in the definition in order to correctly evaluate an expression like:

```
y = SQR(a + b);
```

The preprocessor expands this statement to:

```
y = ((a + b) * (a + b));
```

Without parentheses in the definition, the correct order of evaluation is not preserved, and the preprocessor output is:

```
y = (a + b * a + b);
```

## Notes:

1. Arguments of the **#** and **##** operators are converted *before* replacement of parameters in a function-like macro.
2. The number of arguments in a macro invocation must be the same as the number of parameters in the corresponding macro definition.
3. Commas in the macro invocation argument list do not act as argument separators when they are:
  - in character constants
  - in string literals
  - surrounded by parenthesis
4. The scope of a macro definition begins at the definition and does not end until a corresponding **#undef** directive is encountered. If there is no corresponding **#undef** directive, the scope of the macro lasts until the end of the compilation is reached.
5. A recursive macro is not fully expanded. For example, the definition

```
#define x(a,b) x(a+1,b+1) + 4
```

would expand

```
x(20,10)
```

to

```
x(20+1,10+1) + 4
```

rather than trying to expand the macro *x* over and over within itself.

6. A definition is not required to specify replacement tokens. The following definition removes all instances of the token `debug` from subsequent lines in the current file:

```
#define debug
```

This is the same as specifying the **-Ddebug=** compiler option. Note that specifying **-Ddebug** without the = (equal sign) gives the digit 1 as replacement text.

7. You can change the definition of a defined identifier or macro with a second preprocessor **#define** directive only if the second preprocessor **#define** statement is preceded by a preprocessor **#undef** directive. The **#undef** directive nullifies the first definition so that the same identifier can be used in a redefinition.
8. Within the text of the program, the preprocessor does not scan character constants or string constants for macro invocations.

## Related Reading

“Preprocessor Macros” on page 59

“Preprocessor Directives” on page 58

“Example of the **#define** Preprocessor Directive”

“**#undef** Preprocessor Directive” on page 358

“Predefined Preprocessor Macros” on page 359

“Preprocessor Macro Operators” on page 377

“List of Standard Preprocessor Directives” on page 347

“**#undef** Preprocessor Directive” on page 358

“D” on page 250

“P” on page 307

## Example of the **#define** Preprocessor Directive

The following program contains two macro definitions and a macro invocation that refers to both of the defined macros:

```
/**
 ** This example illustrates #define directives.
 **/
#include <stdio.h>
#define SQR(s) ((s) * (s))
#define PRNT(a,b) \
    printf("value 1 = %d\n", a); \
```

```

    printf("value 2 = %d\n", b) ;
int main(void)
{
    int x = 2;
    int y = 3;
    PRNT(SQR(x),y);
    return(0);
}

```

After being interpreted by the preprocessor, this program is replaced by code equivalent to the following:

```

#include <stdio.h>
int main(void)
{
    int x = 2;
    int y = 3;
    printf("value 1 = %d\n", ( (x) * (x) ) );
    printf("value 2 = %d\n", y);
    return(0);
}

```

This program produces the following output:

```

value 1 = 4
value 2 = 3

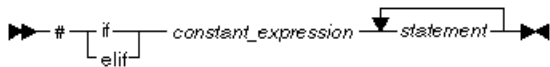
```

### Related Reading

- “Preprocessor Macros” on page 59
- “#define Preprocessor Directive” on page 348
- “List of Standard Preprocessor Directives” on page 347

## #if, #elif Preprocessor Directives

The **#if** and **#elif** directives compare the value of the expression to zero.



If the constant expression evaluates to a nonzero value, the tokens that immediately follow the condition are passed on to the compiler.

If the expression evaluates to zero and the conditional compilation directive contains a preprocessor **#elif** directive, the source text located between the **#elif** and the next **#elif** or **#else** preprocessor directive is selected by the preprocessor to be passed on to the compiler. The **#elif** directive cannot appear after the preprocessor **#else** directive.

All macros are expanded, any defined() expressions are processed and all remaining identifiers are replaced with the token 0.

The expressions that are tested must be integer constant expressions with the following properties:

- No casts are performed.
- Arithmetic is performed using **long int** values.
- The expression can contain defined macros. No other identifiers can appear in the expression.
- The constant expression can contain the unary operator **defined**. This operator can be used only with the preprocessor keyword **#if**. The following expressions evaluate to 1 if the *identifier* is defined in the preprocessor, otherwise to 0:

```

    defined identifier    defined(identifier)

```

For example:

```
#if defined(TEST1) || defined(TEST2)
```

**Note:** If a macro is not defined, a value of 0 (zero) is assigned to it. In the following example, TEST must be a macro identifier:

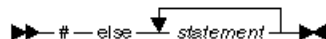
```
#if TEST >= 1
    printf("i = %d\n", i);
    printf("array[i] = %d\n", array[i]);
#elif TEST < 0
    printf("array subscript out of bounds \n");
#endif
```

### Related Reading

“Conditional Compilation Directives” on page 60  
“Preprocessor Directives” on page 58  
“Examples of Conditional Preprocessor Directives” on page 355  
“#else Preprocessor Directive”  
“#endif Preprocessor Directive”  
“#ifdef Preprocessor Directive” on page 354  
“#ifndef Preprocessor Directive” on page 355  
“List of Standard Preprocessor Directives” on page 347

## #else Preprocessor Directive

If the condition specified in the **#if**, **#ifdef**, or **#ifndef** directive evaluates to 0, and the conditional compilation directive contains a preprocessor **#else** directive, the source text located between the preprocessor **#else** directive and the preprocessor **#endif** directive is selected by the preprocessor to be passed on to the compiler.



### Related Reading

“Conditional Compilation Directives” on page 60  
“Preprocessor Directives” on page 58  
“Examples of Conditional Preprocessor Directives” on page 355  
“#if, #elif Preprocessor Directives” on page 352  
“#endif Preprocessor Directive”  
“#if, #elif Preprocessor Directives” on page 352  
“#ifdef Preprocessor Directive” on page 354  
“#ifndef Preprocessor Directive” on page 355  
“List of Standard Preprocessor Directives” on page 347

## #endif Preprocessor Directive

The preprocessor **#endif** directive ends the “**#if, #elif Preprocessor Directives**” on page 352 conditional compilation directive.

▶▶ # — endif ▶▶

### Related Reading

“Conditional Compilation Directives” on page 60  
“Preprocessor Directives” on page 58  
“Examples of Conditional Preprocessor Directives” on page 355  
“#if, #elif Preprocessor Directives” on page 352  
“#else Preprocessor Directive” on page 353  
“#if, #elif Preprocessor Directives” on page 352  
“#ifdef Preprocessor Directive”  
“#ifndef Preprocessor Directive” on page 355  
“List of Standard Preprocessor Directives” on page 347

## #error Preprocessor Directive

A *preprocessor error directive* causes the preprocessor to generate a severe (S) compile-time diagnostic error message. Preprocessing continues, but no object code is generated.

▶▶ # — error — character ▶▶

Use the **#error** directive as a safety check during compilation. For example, if your program uses preprocessor conditional compilation directives, put **#error** directives in the source file to prevent code generation if a section of the program is reached that should be bypassed.

For example, the directive

```
#error Error in TESTPGM1 - This section should not be compiled
```

generates the following error message:

```
Error in TESTPGM1 - This section should not be compiled
```

### Related Reading

“Preprocessor Directives” on page 58  
“List of Standard Preprocessor Directives” on page 347

## #ifdef Preprocessor Directive

The **#ifdef** directive checks for the existence of macro definitions.

▶▶ # — ifdef — identifier — statement ▶▶

If the identifier specified is defined as a macro, the tokens that immediately follow the condition are passed on to the compiler.

The following example defines `MAX_LEN` to be 75 if `EXTENDED` is defined for the preprocessor. Otherwise, `MAX_LEN` is defined to be 50.

```
#ifdef EXTENDED
#   define MAX_LEN 75
#else
#   define MAX_LEN 50
#endif
```

## Related Reading

“Conditional Compilation Directives” on page 60  
“Preprocessor Directives” on page 58  
“Examples of Conditional Preprocessor Directives”  
“#if, #elif Preprocessor Directives” on page 352  
“#endif Preprocessor Directive” on page 353  
“#if, #elif Preprocessor Directives” on page 352  
“#ifndef Preprocessor Directive”  
“List of Standard Preprocessor Directives” on page 347

## #ifndef Preprocessor Directive

The **#ifndef** directive checks for the existence of macro definitions.

```
▶▶ # ifndef identifier statement ▶▶
```

If the identifier specified is not defined as a macro, the tokens that immediately follow the condition are passed on to the compiler.

An identifier must follow the **#ifndef** keyword. The following example defines `MAX_LEN` to be 50 if `EXTENDED` is not defined for the preprocessor. Otherwise, `MAX_LEN` is defined to be 75.

```
#ifndef EXTENDED
#   define MAX_LEN 50
#else
#   define MAX_LEN 75
#endif
```

## Related Reading

“Conditional Compilation Directives” on page 60  
“Preprocessor Directives” on page 58  
“Examples of Conditional Preprocessor Directives”  
“#if, #elif Preprocessor Directives” on page 352  
“#endif Preprocessor Directive” on page 353  
“#if, #elif Preprocessor Directives” on page 352  
“#ifdef Preprocessor Directive” on page 354  
“List of Standard Preprocessor Directives” on page 347

## Examples of Conditional Preprocessor Directives

### Example 1

The following example shows how you can nest preprocessor conditional compilation directives:

```
#if defined(TARGET1)
#   define SIZEOF_INT 16
#   ifdef PHASE2
#       define MAX_PHASE 2
#   else
#       define MAX_PHASE 8
#   endif
#elif defined(TARGET2)
#   define SIZEOF_INT 32
#   define MAX_PHASE 16
#else
#   define SIZEOF_INT 32
#   define MAX_PHASE 32
#endif
```

## Example 2

The following program contains preprocessor conditional compilation directives:

```
/**
 ** This example contains preprocessor
 ** conditional compilation directives.
 **/
#include <stdio.h>
int main(void)
{
    static int array[ ] = { 1, 2, 3, 4, 5 };
    int i;
    for (i = 0; i <= 4; i++)
    {
        array[i] *= 2;
    }
    #if TEST >= 1
        printf("i = %d\n", i);
        printf("array[i] = %d\n", array[i]);
    #endif
    }
    return(0);
}
```

## Related Reading

“Conditional Compilation Directives” on page 60

“#if, #elif Preprocessor Directives” on page 352

“#else Preprocessor Directive” on page 353

“#endif Preprocessor Directive” on page 353

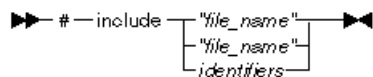
“#if, #elif Preprocessor Directives” on page 352

“#ifdef Preprocessor Directive” on page 354

“#ifndef Preprocessor Directive” on page 355

## #include Preprocessor Directive

A *preprocessor include directive* causes the preprocessor to replace the directive with the contents of the specified file.



The preprocessor resolves macros contained in a **#include** directive. After macro replacement, the resulting token sequence must consist of a file name enclosed in either double quotation marks or the characters < and >. For example:

```
#define MONTH <july.h>
#include MONTH
```

If the file name is enclosed in double quotation marks, (“) the preprocessor searches the place (for example, directories or libraries) that contain the source files and then a standard or specified sequence of places until it finds the specified file. For example:

```
#include "payroll.h"
```

If the file name is enclosed in the characters < and >, the preprocessor searches only the standard or specified places for the specified file. For example:

```
#include <stdio.h>
```

The -I compiler option specifies a search path if the file name in the **#include** directive is not an absolute path.



## Related Reading

“Preprocessor Directives” on page 58  
“Examples of the #include Preprocessor Directive”  
“List of Standard Preprocessor Directives” on page 347  
“l” on page 272

## Examples of the #include Preprocessor Directive

### Example 1

Declarations that are used by several files can be placed in one file and included with **#include** in each file that uses them. For example, the following file `defs.h` contains several definitions and an inclusion of an additional file of declarations:

```
/* defs.h */
#define TRUE 1
#define FALSE 0
#define BUFFERSIZE 512
#define MAX_ROW 66
#define MAX_COLUMN 80
int hour;
int min;
int sec;
#include "mydefs.h"
```

You can embed the definitions that appear in `defs.h` with the following directive:

```
#include "defs.h"
```

The preprocessor looks for the file **defs.h** first in the directory that contains the source file. If the file is not found there, the preprocessor searches a sequence of specified or standard locations.

### Example 2

In the following example, a **#define** combines several preprocessor macros to define a macro that represents the name of the C standard I/O header file. A **#include** makes the header file available to the program.

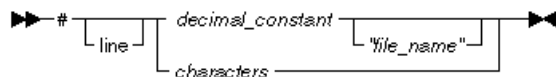
```
#define IO_HEADER <stdio.h>
:
:
:
#include IO_HEADER /* equivalent to specifying #include <stdio.h> */
:
:
:
```

## Related Reading

“Preprocessor Directives” on page 58  
“#include Preprocessor Directive” on page 356  
“List of Standard Preprocessor Directives” on page 347

## #line Preprocessor Directive

A *preprocessor line control directive* supplies line numbers for compiler messages. It causes the compiler to view the line number of the next source line as the specified number.



In order for the compiler to produce meaningful references to line numbers in preprocessed source, the preprocessor inserts **#line** directives where necessary (for example, at the beginning and after the end of included text).

A file name specification enclosed in double quotation marks can follow the line number. If you specify a file name, the compiler views the next line as part of the specified file. If you do not specify a file name, the compiler views the next line as part of the current source file.

The token sequence on a **#line** directive is subject to macro replacement. After macro replacement, the resulting character sequence must consist of a decimal constant, optionally followed by a file name enclosed in double quotation marks.

**Note:** In **extended** mode, the keyword **line** is optional. The directive

```
# line 300
```

is equivalent to

```
# 300
```

The keyword **line** is required in **ansi** mode.

### Related Reading

“Preprocessor Directives” on page 58

“List of Standard Preprocessor Directives” on page 347

## #undef Preprocessor Directive

A *preprocessor undef directive* causes the preprocessor to end the scope of a preprocessor definition.

```
▶▶ # undef identifier ▶▶
```

If the identifier is not currently defined as a macro, **#undef** is ignored

Macros can also be undefined with the **-U** compiler option.

### Example of Usage

The following directives define **BUFFER** and **SQR**:

```
#define BUFFER 512
#define SQR(x) ((x) * (x))
```

The following directives nullify these definitions:

```
#undef BUFFER
#undef SQR
```

Any occurrences of the identifiers **BUFFER** and **SQR** that follow these **#undef** directives are not replaced with any replacement tokens. Once the definition of a macro has been removed by an **#undef** directive, the identifier can be used in a new **#define** directive.

### Related Reading

“Preprocessor Macros” on page 59

“Preprocessor Directives” on page 58

“#define Preprocessor Directive” on page 348

“List of Standard Preprocessor Directives” on page 347

“#define Preprocessor Directive” on page 348

“U” on page 332

---

## Predefined Preprocessor Macros

C for AIX includes two groups of predefined preprocessor macros. The first group contains macros defined by the ANSI standard for the C programming language. The second group contains macros provided by C for AIX.

### ANSI Standard Predefined Preprocessor Macros

| Name                  | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__LINE__</code> | <p>An integer describing the current source line number.</p> <p>The value of <code>__LINE__</code> changes during compilation as the compiler processes subsequent lines of your source program. It can be set with the <code>#line</code> directive.</p>                                                                                                                                                                                                                                                                                                                                                                |
| <code>__FILE__</code> | <p>A character string literal containing the name of the source file.</p> <p>The value of <code>__FILE__</code> changes as the compiler processes include files that are part of your source program. It can be set with the <code>#line</code> directive.</p>                                                                                                                                                                                                                                                                                                                                                           |
| <code>__DATE__</code> | <p>A character string literal containing the date when the source file was compiled.</p> <p>The value of <code>__DATE__</code> changes as the compiler processes any include files that are part of your source program. The date is in the form:</p> <p style="padding-left: 40px;"><code>"Mmm dd yyyy"</code></p> <p>where:</p> <p><b>Mmm</b> Represents the month in an abbreviated form (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec).</p> <p><b>dd</b> Represents the day of the month. If the day is less than 10, the first d is a blank character.</p> <p><b>yyyy</b> Represents the year.</p> |
| <code>__STDC__</code> | <p>The integer 1 (one) indicates that the C compiler conforms to the ANSI standard.</p> <p><b>Note:</b> This macro is undefined if the language level is set to anything other than ANSI.</p>                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <code>__TIME__</code> | <p>A character string literal containing the time when the source file was compiled.</p> <p>The value of <code>__TIME__</code> changes as the compiler processes any include files that are part of your source program. The time is in the form:</p> <p style="padding-left: 40px;"><code>"hh:mm:ss"</code></p> <p>where:</p> <p><b>hh</b> Represents the hour.</p> <p><b>dd</b> Represents the minutes.</p> <p><b>yyyy</b> Represents the seconds.</p> <p>The time is always set to the system time.</p>                                                                                                               |

| Name                       | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__TIMESTAMP__</code> | <p>A character string literal containing the date and time when the source file was last modified.</p> <p>The value of <code>__TIMESTAMP__</code> changes as the compiler process any include files that are part of your source program. The date and the time are in the form:</p> <pre>"Day Mmm dd hh:mm:ss yyyy"</pre> <p>where:</p> <p><b>Day</b> Represents the day of the week. (Mon, Tue, Wed, Thu, Fri, Sat, or Sun).</p> <p><b>Mmm</b> Represents the month in an abbreviated form (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec).</p> <p><b>dd</b> Represents the day of the month. If the day is less than 10, the first d is a blank character.</p> <p><b>hh</b> Represents the hour.</p> <p><b>mm</b> Represents the minutes.</p> <p><b>ss</b> Represents the seconds.</p> <p><b>yyyy</b> Represents the seconds.</p> <p>The date and time are always set to the system date and time.</p> |

**Note:** You cannot use the **-U** option to undefine a predefined macro name.

### C for AIX Predefined Preprocessor Macros

| Name                        | Description                                                                                                                                                                                   |
|-----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__64BIT__</code>      | Defined if the compiler is invoked to compile in 64-bit mode. This macro should not be user-defined or redefined.                                                                             |
| <code>_AIX32</code>         | Defined if the operating system is AIX version 3.2 or higher.                                                                                                                                 |
| <code>_AIX41</code>         | Defined if the operating system is AIX version 4.1 or higher.                                                                                                                                 |
| <code>_AIX43</code>         | Defined if the operating system is AIX version 4.3 or higher.                                                                                                                                 |
| <code>__ANSI__</code>       | Allows only language constructs that conform to ANSI C standards.<br>Defined using the <b>#pragma langlvl</b> directive or the <b>-qlanglvl</b> compiler option.                              |
| <code>_ARCH_*</code>        | Indicates that the compiler generates code to run on the family of processors denoted by *.<br>See the <b>-qarch</b> compiler option for more information.                                    |
| <code>_CHAR_SIGNED</code>   | Indicates that the default character type is signed.<br>Defined when the <b>-qchars=signed</b> compiler option is in effect. See the <b>-qchars</b> compiler option for more information.     |
| <code>_CHAR_UNSIGNED</code> | Indicates that the default character type is unsigned.<br>Defined when the <b>-qchars=unsigned</b> compiler option is in effect. See the <b>-qchars</b> compiler option for more information. |
| <code>__CLASSIC__</code>    | Macro defined when the <b>classic</b> language level is specified.<br>Defined using the <b>#pragma langlvl</b> directive or the <b>-qlanglvl</b> compiler option.                             |
| <code>__EXTENDED__</code>   | Allows additional language constructs provided by the C for AIX implementation.<br>Defined using the <b>#pragma langlvl</b> directive or the <b>-qlanglvl</b> compiler option.                |

| Name                        | Description                                                                                                                                                                                                                                                                                                                           |
|-----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__FUNCTION__</code>   | Indicates the name of the function being compiled.                                                                                                                                                                                                                                                                                    |
| <code>__HOS_AIX__</code>    | Indicates the host operating system is AIX.                                                                                                                                                                                                                                                                                           |
| <code>__IBMC__</code>       | Macro contains the version number of the compiler, for example, <code>__IBMC__=450</code> . This macro should be used in new code.                                                                                                                                                                                                    |
| <code>__IBMSMP</code>       | Macro defined when the when the <code>-qsmp</code> compiler option is selected.                                                                                                                                                                                                                                                       |
| <code>_ILP32</code>         | Defined if the compiler is using the 32-bit data model. This data model is used when compiling programs for 32-bit mode. This macro should not be user-defined or redefined.                                                                                                                                                          |
| <code>_LONG_LONG</code>     | Macro defined when the compiler is in a mode that permits <b>long long int</b> and <b>unsigned long long int</b> types.                                                                                                                                                                                                               |
| <code>_LONGDOUBLE128</code> | Sets the number of bits to use when representing the value of a long double. The available options are 64 and 128 bits.                                                                                                                                                                                                               |
| <code>_LP64</code>          | Defined if the compiler is using the 64-bit data model. This data model is used when compiling programs for 64-bit mode. This macro should not be user-defined or redefined.                                                                                                                                                          |
| <code>__MATH__</code>       | Instructs the compiler to generate substitute code for calls to some math functions available in the standard C runtime libraries, if appropriate.<br><br>The functions handled this way are defined as replacement text for macros that begin with two underscores (__) in the <code>/usr/vac/include/math.h</code> header file.     |
| <code>_OPENMP</code>        | Macro defined when the <code>-qsmp=omp</code> compiler option is set to enable full compliance to the OpenMP API specification.                                                                                                                                                                                                       |
| <code>_POWER</code>         | Indicates the operating system is AIX 4.1 or higher.                                                                                                                                                                                                                                                                                  |
| <code>__SAA__</code>        | Allows only language constructs that conform to the most recent level of the SAA C standards.<br><br>Defined using the <code>#pragma langlvl</code> directive or the <code>-qlanglvl</code> compiler option.                                                                                                                          |
| <code>__SAAL2__</code>      | Allows only language constructs that conform to the most recent level of the SAA Level 2 C standards.<br><br>Defined using the <code>#pragma langlvl</code> directive or the <code>-qlanglvl</code> compiler option.                                                                                                                  |
| <code>__STR__</code>        | Instructs the compiler to generate substitute code for calls to some string functions available in the standard C runtime libraries, if appropriate.<br><br>The functions handled this way are defined as replacement text for macros that begin with two underscores (__) in the <code>/usr/vac/include/string.h</code> header file. |
| <code>__THW_INTEL__</code>  | Indicates that the target hardware is an Intel processor.                                                                                                                                                                                                                                                                             |
| <code>__THW_RS6000__</code> | Indicates that the target hardware is a RISC/6000 processor.                                                                                                                                                                                                                                                                          |
| <code>__x1C__</code>        | A hexadecimal constant containing the version number of the compiler. The version is in the form:<br><br>0xVVRR<br><br>where:<br><br><b>VV</b> Represents the compiler version number.<br><b>RR</b> Represents the compiler release number.<br><br>For C for AIX Version 5 Release 0, the macro has the value 0x0500.                 |

| Name                    | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__XLC121__</code> | <p>Instructs the compiler to generate substitute code for calls to some new string and math functions.</p> <p>The functions handled this way are defined as replacement text for macros that begin with two underscores (<code>__</code>) in the following header files:</p> <ul style="list-style-type: none"> <li>• <code>/usr/vac/include/string.h</code></li> <li>• <code>/usr/vac/include/math.h</code></li> <li>• <code>/usr/vac/include/stdlib.h</code></li> <li>• <code>/usr/vac/include/stream.h</code></li> </ul> |

### Notes:

1. The value of all C for AIX macros are defined when the corresponding **#pragma** directive or compiler option is used.
2. Except for `__MATH`, `__STR`, and `__XLC121` macros, predefined macro names cannot be the subject of a **#define** or **#undefine** preprocessor directive. The preprocessor ignores any redefined macros and issues an error message.
3. You cannot use the **-U** option to undefine a predefined macro name.

### Related Reading

"Preprocessor Directives" on page 58  
 "Preprocessor Macros" on page 59  
 "C Language Levels" on page 78  
 "Examples of Predefined Macros in a Program"  
 "#define Preprocessor Directive" on page 348  
 "#line Preprocessor Directive" on page 357  
 "#pragma langlvl Preprocessor Directive" on page 373  
 "Preprocessor Macro Operators" on page 377  
 "List of Standard Preprocessor Directives" on page 347  
 "U" on page 332  
 "arch" on page 237  
 "langlvl" on page 286  
 "smp" on page 320

## Examples of Predefined Macros in a Program

### Example 1

The following `printf` statements display the values of the predefined macros `__LINE__`, `__FILE__`, `__TIME__`, and `__DATE__` and print a message indicating the program's conformance to ANSI/ISO standards based on `__STDC__`:

```

/**
 ** This example illustrates some predefined macros.
 **/
#pragma langlvl(ANSI)
#include <stdio.h>
#if __STDC__
#   define CONFORM    "conforms"
#else
#   define CONFORM    "does not conform"
#endif
int main(void)
{
    printf("Line %d of file %s has been executed\n", __LINE__, __FILE__);
    printf("This file was compiled at %s on %s\n", __TIME__, __DATE__);
    printf("This program %s to ANSI/ISO standard C\n", CONFORM);
}

```

## Example 2

The following program uses the `__FUNCTION__` macro to determine the name of the program function currently in effect.

```
/**
 ** This example illustrates the __FUNCTION__ predefined macro
 ** in a C program.
 **/
#include <stdio.h>
int foo(int);
main(int argc, char **argv) {
    int k = 1;
    printf (" In function %s \n",__FUNCTION__);
    foo(k);
}
int foo (int i) {
    printf (" In function %s \n",__FUNCTION__);
}
```

The output of this example is:

```
    In function main
    In function foo
```

## Related Reading

- “Preprocessor Directives” on page 58
- “Predefined Preprocessor Macros” on page 359
- “#define Preprocessor Directive” on page 348
- “#line Preprocessor Directive” on page 357
- “#undef Preprocessor Directive” on page 358
- “List of Standard Preprocessor Directives” on page 347

---

## #pragma Preprocessor Directives

A *pragma* is an implementation-defined instruction to the compiler. It has the general form:

```
▶▶ # pragma character_sequence ▶▶
```

where *character\_sequence* is a series of characters giving a specific compiler instruction and arguments, if any.

The *character\_sequence* on a `pragma` is not subject to macro substitutions. More than one `pragma` construct can be specified on a single **#pragma** directive. The compiler ignores unrecognized `pragmas`.

Some **#pragma** directives, as indicated in the list below, must appear before any statements in the C source code. The other **#pragma** directives can be used throughout your program to affect a selected block of source code.

The C for AIX compiler lets you specify many compiler options as either command line options or as **#pragma** statements. In addition, the C for AIX compiler recognizes the `pragmas` listed below:

| Pragma Directive | Description |
|------------------|-------------|
|------------------|-------------|

|                                                                           |                                                                                                                                                |
|---------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| “#pragma alloca<br>Preprocessor<br>Directive” on page 365                 | Provides an inline version of function <b>alloca</b> . This directive must appear before any statements in the C source code.                  |
| “#pragma chars<br>Preprocessor<br>Directive” on page 365                  | Sets the sign type of character data. This directive must appear before any statements in the C source code.                                   |
| “#pragma comment<br>Preprocessor<br>Directive” on page 366                | Places a comment into the object file.                                                                                                         |
| “#pragma disjoint<br>Preprocessor<br>Directive” on page 366               | Lists identifiers not aliased to each other within the current scope of their use.                                                             |
| “#pragma<br>execution_frequency<br>Preprocessor<br>Directive” on page 367 | Identifies the expected frequency with which a block of code will be executed.                                                                 |
| “#pragma hdrfile<br>Preprocessor<br>Directive” on page 368                | Specifies the file name of the precompiled header to be generated and/or used.                                                                 |
| “#pragma hdrstop<br>Preprocessor<br>Directive” on page 369                | Terminates the initial sequence of #include directives being considered for precompilation.                                                    |
| “#pragma info<br>Preprocessor<br>Directive” on page 370                   | Controls the diagnostic messages generated by the <b>-qinfo</b> compiler option.                                                               |
| “#pragma isolated_call<br>Preprocessor<br>Directive” on page 371          | Lists functions that do not alter data objects visible at the time of the function call.                                                       |
| “#pragma langlvl<br>Preprocessor<br>Directive” on page 373                | Selects the C language level for compilation. This directive must appear before any statements in the C source code.                           |
| “#pragma leaves<br>Preprocessor<br>Directive” on page 373                 | Specifies that a given function never returns.                                                                                                 |
| “#pragma map<br>Preprocessor<br>Directive” on page 374                    | Tells the compiler that all references to an identifier are to be converted to “name”.                                                         |
| “#pragma<br>option_override<br>Preprocessor<br>Directive” on page 374     | Lets you specify alternate optimization options for specific functions.                                                                        |
| “#pragma options<br>Preprocessor<br>Directive” on page 375                | Specifies settings for compiler options in your source program.                                                                                |
| “#pragma reachable<br>Preprocessor<br>Directive” on page 376              | Specifies that the point after a given routine, marked <i>reachable</i> , can be reached from a point other than the return from that routine. |
| “#pragma strings<br>Preprocessor<br>Directive” on page 376                | Sets storage type for strings. This directive must appear before any statements in the C source code.                                          |

**Note:** The **#pragma page**, **#pragma skip**, **#pragma subtitle**, and **#pragma title** directives are not recognized by the C for AIX compiler.



Another set of pragma directives let you control parallel processing operations. See “#pragma Preprocessor Directives for Parallel Processing” on page 381 for more information.

#### Related Reading

“Preprocessor Directives” on page 58

“List of Standard Preprocessor Directives” on page 347

“#pragma Preprocessor Directives for Parallel Processing” on page 381

“Compiler Options and Their Defaults” on page 218

“info” on page 275

## #pragma alloca Preprocessor Directive

The **#pragma alloca** directive specifies that the function **alloca(size\_t size)** is to allocate space for an object of *size* bytes. The allocated space is put on the stack.

▶ # — pragma — alloca — ▶

You must include the **#pragma alloca** directive to have the compiler provide an inline version of **alloca**. Alternatively, the **-ma** compiler option substitutes inline code for calls to function **alloca** without specifying the **#pragma alloca** directive in the source code. 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 pragma must be included in the source before the first function definition. Once specified, it applies to the rest of the file and cannot be turned off. If a program source contains functions that you want compiled without **#pragma alloca**, place these functions in a different file.

Whenever you make a call to **alloca**, you must include the header file **<malloc.h>** to define **alloca**. Header files are described in the *AIX Version 4 Files Reference*.

#### Related Reading

“Preprocessor Directives” on page 58

“#pragma Preprocessor Directives” on page 363

“List of Standard Preprocessor Directives” on page 347

“ma” on page 295

## #pragma chars Preprocessor Directive

The **#pragma chars** directive specifies that the compiler is to treat all **char** objects as **signed** or **unsigned**.

▶ # — pragma — chars — ( — signed — ) —  
                                        |  
                                        | — unsigned — |

This pragma must appear before any statements in a file. Once specified, it applies to the rest of the file and cannot be turned off. If a program file contains functions that you want compiled without **#pragma chars**, place these functions in a different file.

The **chars** compiler option has the same effect as this pragma.

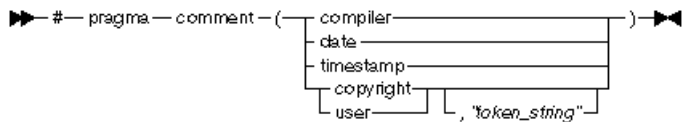
The **\_CHAR\_SIGNED** or **\_CHAR\_UNSIGNED** macros are defined according to the setting of the **-qchars** option or corresponding preprocessor directives.

## Related Reading

- “Preprocessor Directives” on page 58
- “#pragma Preprocessor Directives” on page 363
- “List of Standard Preprocessor Directives” on page 347
- “chars” on page 244

## #pragma comment Preprocessor Directive

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



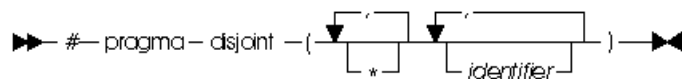
|           |                                                                                                                                                                   |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| compiler  | The compiler appends the name and version of the compiler to the end of the generated object module.                                                              |
| date      | The compiler appends the date and time of compilation to the end of the generated object module.                                                                  |
| timestamp | The compiler appends the date and time of the last modification to the sourcer to the end of the generated object module.                                         |
| copyright | The compiler places text specified by the <i>token_string</i> into the generated object module. This text loads into memory when the program runs.                |
| user      | The compiler places text specified by the <i>token_string</i> into the generated object module. This text does <i>not</i> load into memory when the program runs. |

## Related Reading

- “Preprocessor Directives” on page 58
- “#pragma Preprocessor Directives” on page 363
- “List of Standard Preprocessor Directives” on page 347

## #pragma disjoint Preprocessor Directive

The **#pragma disjoint** directive lists the identifiers that are not aliased to each other within the scope of their use.



where *identifier* is a primary expression that can be the name of an operator function, conversion function, destructor, or a qualified name.

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 give incorrect results.

The pragma can appear anywhere in the source program that a declaration is allowed. 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

The identifiers must be declared before they are used in the pragma. A pointer in the identifier list must not have been dereferenced or used as a function argument before appearing in the directive.

The **-qignprag** compiler option causes aliasing pragmas to be ignored. Use this option to debug applications containing the **#pragma disjoint** directive.

#### Related Reading

“Preprocessor Directives” on page 58

“Example of the #pragma disjoint Preprocessor Directive”

“#pragma Preprocessor Directives” on page 363

“List of Standard Preprocessor Directives” on page 347

“ignprag” on page 274

“ignprag” on page 274

### Example of the #pragma disjoint Preprocessor Directive

The following example shows the use of **#pragma disjoint**.

```
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 will not change 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 to `another_function` has the value 6 and will not reload the variable from memory.

#### Related Reading

“Preprocessor Directives” on page 58

“#pragma disjoint Preprocessor Directive” on page 366

“#pragma Preprocessor Directives” on page 363

“List of Standard Preprocessor Directives” on page 347

### #pragma execution\_frequency Preprocessor Directive

The **#pragma execution\_frequency** directive identifies the expected frequency with which a block of code will be executed. This information is used by the compiler as hint to the optimizer.

▶▶ # — pragma — execution\_frequency — ( — frequency — ) —▶▶

The currently accepted value for *frequency* is:

**very\_low**

The probability of execution for the statement block in which the pragma resides is very close to nil.

This pragma has effect only if:

- the program is optimized.
- the pragma is placed inside statements with block scope such as if-then-else, looping and switch statements. A warning message is issued and the pragma ignored if it is placed outside of block scope.

## Examples

1. In the following sample program, execution is unlikely to branch through Block A:

```
int *array = (int *) malloc(10000);
if (array == NULL) {
    /* Block A */
    #pragma execution_frequency(very_low)    error();}
```

2. In the following sample program, code “Block B” is marked as being infrequently executed, indicating that “Block C” is most likely to be chosen during branching.

```
if (Foo > 0) {    #pragma execution_frequency(very_low)
    /* Block B */
    doSomething();
} else {
    /* Block C */
    doAnotherThing();
}
```

## Related Reading

“Preprocessor Directives” on page 58

“List of Standard Preprocessor Directives” on page 347

## #pragma hdrfile Preprocessor Directive

The *#pragma hdrfile* directive specifies the file name of the precompiled header to be generated and/or used.

▶▶ #— pragma — hdrfile — filename —▶▶

This pragma must appear before the first *#include* directive, and either the **-qgenpcomp** or **-qusepcomp** compiler options must also be specified.

If a file name is specified by both a **-qgenpcomp** or **-qusepcomp** compiler option and a **#pragma hdrfile** entry, the name specified by the pragma takes precedence. If the name specified is a directory, the compiler searches for or generates a file with the default name in that directory.

In order to maximize the reuse of precompiled headers, use **#pragma hdrfile** in combination with **#pragma hdrstop** to manually limit the initial sequence of *#include* directives.

Using precompiled header files can decrease compile time. Using precompiled headers will not improve compile time performance in most applications without some organization of the headers included by each source file.

Some examples of **#pragma hdrfile** directives are:

```
/******
 *
 * In the following example, the headers h1.h and h2.h are precompiled and
 * the precompiled output is written to the file fred.pch (provided the
 * -qgenpcomp compiler option is specified). If -qgenpcomp=dave.pch is
 * specified, the precompiled output will still be written to fred.pch since
 * the name specified in the pragma takes precedence. To use the precompiled
 * output in fred.pch when compiling, specify the -qusepcomp compiler option.
 *
 */
```

```

*****/
#pragma hdrfile "fred.pch"
#include "h1.h"
#include "h2.h"
main () {}
/*****
*
In the following example, only header h1.h will be precompiled (provided
the -qgenpcomp compiler option is specified) and the precompiled output
is written to the file fred.pch. To use the precompiled output in fred.pch
when compiling, specify the -qusepcomp compiler option.
*
*****/
#pragma hdrfile "fred.pch"
#include "h1.h"
#pragma hdrstop
#include "h2.h"
main () {}

```

### Related Reading

- “Preprocessor Directives” on page 58
- “List of Standard Preprocessor Directives” on page 347
- “#pragma hdrstop Preprocessor Directive”
- “genpcomp” on page 267
- “usepcomp” on page 335

## #pragma hdrstop Preprocessor Directive

The *#pragma hdrstop* directive manually terminates the initial sequence of *#include* directives being considered for precompilation.

▶▶ # — pragma ——— hdrstop —▶▶

It has no effect if:

- The initial sequence of *#include* directives has already ended
- Neither the **-qgenpcomp** or **-qusepcomp** compiler options are specified
- It does not appear in the primary source file

Using precompiled header files can decrease compile time. Using precompiled headers will not improve compile time performance in most applications without some organization of the headers included by each source file.

Some examples of **#pragma hdrfile** directives are:

```

/*****
*
In the following example, only header file h1.h is precompiled and the
precompiled output is written to the file csetc.pch (provided the
-qgenpcomp compiler option is specified). If both -qusepcomp=dave.pch
and -qgenpcomp=john.pch are specified then the compiler looks for the
precompiled header in john.pch (since this is the name specified last),
and regenerates it if it is not found or unusable.
*
*****/
#include "h1.h"
#pragma hdrstop
#include "h2.h"
main () {}
/*****
*

```

In the following example, no precompiled headers are generated or used for the compilation, even if `-qgenpcomp` or `-qusepcomp` compiler options are specified.

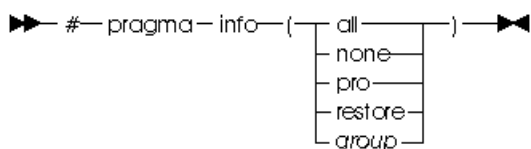
```
*
*****/
#pragma hdrstop
#include "h1.h"
#include "h2.h"
main () {}
```

### Related Reading

- “Preprocessor Directives” on page 58
- “List of Standard Preprocessor Directives” on page 347
- “genpcomp” on page 267
- “usepcomp” on page 335

## #pragma info Preprocessor Directive

The `#pragma info` directive controls the diagnostic messages generated by the `info` compiler option.



You can use this directive in place of the `info` option to turn groups of diagnostic messages on or off. The `#pragma info` directive overrides any `info` options stated on the command line.

Available options are:

- |                      |                                                                                                   |
|----------------------|---------------------------------------------------------------------------------------------------|
| <code>all</code>     | Turns on all diagnostic checking.                                                                 |
| <code>none</code>    | Turns off all diagnostic suboptions for specific portions of your program.                        |
| <code>restore</code> | Restores the options that were in effect before the previous <code>#pragma info</code> directive. |

Because `#pragma info` operates like a stack, the options restored may not be those given on the command line. If no options were previously in effect, `#pragma info(restore)` does nothing.

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

| <b>group</b> | <b>Type of messages returned</b>                             |
|--------------|--------------------------------------------------------------|
| <b>cmp</b>   | Possible redundancies in unsigned comparisons                |
| <b>cnd</b>   | Possible redundancies or problems in conditional expressions |
| <b>cns</b>   | Operations involving constants                               |
| <b>cnv</b>   | Conversions                                                  |
| <b>dc1</b>   | Consistency of declarations                                  |
| <b>eff</b>   | Statements with no effect                                    |
| <b>enu</b>   | Consistency of enum variables                                |
| <b>ext</b>   | Unused external definitions                                  |
| <b>gen</b>   | General diagnostic messages                                  |
| <b>gnr</b>   | Generation of temporary variables                            |
| <b>got</b>   | Use of goto statements                                       |
| <b>ini</b>   | Possible problems with initialization                        |
| <b>inl</b>   | Functions not inlined                                        |
| <b>lan</b>   | Language level effects                                       |
| <b>obs</b>   | Obsolete features                                            |
| <b>ord</b>   | Unspecified order of evaluation                              |
| <b>par</b>   | Unused parameters                                            |
| <b>por</b>   | Nonportable language constructs                              |
| <b>ppc</b>   | Possible problems with using the preprocessor                |
| <b>ppt</b>   | Trace of preprocessor actions                                |
| <b>pro</b>   | Missing function prototypes                                  |
| <b>rea</b>   | Code that cannot be reached                                  |
| <b>ret</b>   | Consistency of return statements                             |
| <b>trd</b>   | Possible truncation or loss of data or precision             |
| <b>tru</b>   | Variable names truncated by the compiler                     |
| <b>uni</b>   | Uninitialized variables                                      |
| <b>use</b>   | Unused auto and static variables                             |
| <b>vft</b>   | Generation of virtual function tables                        |

### **Related Reading**

“Preprocessor Directives” on page 58

“#pragma Preprocessor Directives” on page 363

“List of Standard Preprocessor Directives” on page 347

“info” on page 275

## **#pragma isolated\_call Preprocessor Directive**

The **#pragma isolated\_call** directive lists functions that do not alter data objects visible at the time of the function call.

▶▶ # pragma isolated\_call ( identifier ) ▶▶

The pragma must appear before calls to the functions in the identifier list. The identifiers listed must be declared before they are used in the pragma, and must be of type function or a **typedef** of function.

The pragma informs the compiler that none of the functions listed has side effects. Functions are considered to have side effects if they:

- Access a volatile object
- Modify an external object
- Modify a file
- Call a function that does any of the above.

Any change in the state of the runtime environment is considered a side effect. Passing function arguments by reference is one side effect that is allowed, but in general, functions with side effects can give incorrect results when listed in **#pragma isolated\_call** directives.

Marking a function as isolated indicates to the optimizer that external and static variables cannot be changed by the called function, and that 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. Note that instruction reordering might yield code with more values in general purpose and/or floating-point registers maintained across the isolated call. When the isolated call is not located in a loop, the overhead of saving and restoring extra registers might not be worth the savings that result from deleting the storage references.

Functions specified in the identifier are permitted to examine external objects and return results that depend on the state of the runtime environment. The functions can also modify the storage pointed to by any pointer arguments passed on 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 **-qisolated\_call** compiler option has the same effect as this pragma. The **-qignprag** compiler option causes aliasing programs to be ignored. Use this option to debug applications containing the **#pragma isolated\_call** directive.

### Related Reading

- “Preprocessor Directives” on page 58
- “Example of the #pragma isolated\_call Preprocessor Directive”
- “#pragma Preprocessor Directives” on page 363
- “List of Standard Preprocessor Directives” on page 347
- “ignprag” on page 274
- “isolated\_call” on page 284

## Example of the #pragma isolated\_call Preprocessor Directive

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 argument to `that_function` has the value 6.

```
int a, this_function(int) /* Assumed to have no side effects */
#pragma isolated_call(this_function)
that_function()
{
    a = 6;
    this_function(7); /* Call does not change the value of "a" */
    other_function(a); /* Argument "a" has the value of 6 */
}
```

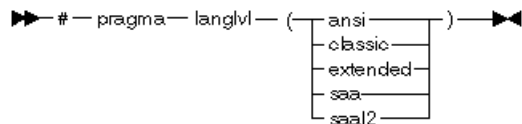


### Related Reading

- “Preprocessor Directives” on page 58
- “#pragma isolated\_call Preprocessor Directive” on page 371
- “#pragma Preprocessor Directives” on page 363
- “List of Standard Preprocessor Directives” on page 347

## #pragma langlvl Preprocessor Directive

The **#pragma langlvl** directive selects the C language level used for compilation.



This pragma 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.

Language levels available with the C for AIX compiler are:

| Language Level | Description                                                                                                                                                                                                        |
|----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ansi           | Defines the predefined macros <b>__ANSI__</b> and <b>__STDC__</b> , and defines other <b>langlvl</b> variables. The default language level for the <b>clc</b> and <b>c89</b> compiler invocations is <b>ansi</b> . |
| classic        | Defines the predefined macro <b>__CLASSIC__</b> , and undefines other <b>langlvl</b> variables.                                                                                                                    |
| extended       | Defines the predefined macro <b>__EXTENDED__</b> , and undefines other <b>langlvl</b> variables. The default language level for the <b>CC</b> compiler invocation commands is <b>extended</b> .                    |
| saa            | Defines the predefined macro <b>__SAA__</b> , and undefines other <b>langlvl</b> variables.                                                                                                                        |
| saa12          | Defines the predefined macro <b>__SAA_L2__</b> , and undefines other <b>langlvl</b> variables.                                                                                                                     |

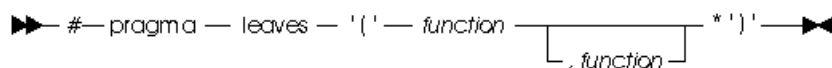
This pragma has the same effect as the **-qlanglvl** compiler option.

### Related Reading

- “Preprocessor Directives” on page 58
- “C Language Levels” on page 78
- “#pragma Preprocessor Directives” on page 363
- “List of Standard Preprocessor Directives” on page 347
- “langlvl” on page 286

## #pragma leaves Preprocessor Directive

The **#pragma leaves** directive takes a *function* name, and specifies that the function never returns to the instruction following that function call.



If the specified function is not found, a warning message is produced.

#### Related Reading

“Preprocessor Directives” on page 58

“C Language Levels” on page 78

“#pragma Preprocessor Directives” on page 363

“List of Standard Preprocessor Directives” on page 347

## #pragma map Preprocessor Directive

The *#pragma map* directive tells the compiler that all references to an function identifier are to be converted to “*name*”.

```
▶▶ # - pragma — map — ( — identifier — "name" — ) —▶▶
```

The following describes the options available for **#pragma map**:

|                   |                                                          |
|-------------------|----------------------------------------------------------|
| <i>identifier</i> | Name of a function.                                      |
| <i>name</i>       | External name that is to be bound to the given function. |

The directive can appear anywhere in the program. The identifier appearing in the directive is resolved as though the directive had appeared at file scope, independent of its actual point of occurrence.

For example:

```
int func(int);
{
    void func(void);
#pragma map(func, "funcname1") /* maps func to funcname1 */
};
```

#### Related Reading

“Preprocessor Directives” on page 58

“Example of the #pragma isolated\_call Preprocessor Directive” on page 372

“#pragma Preprocessor Directives” on page 363

“List of Standard Preprocessor Directives” on page 347

## #pragma option\_override Preprocessor Directive

The **#pragma option\_override** directives lets you specify alternate optimization options for specific functions.

```
▶▶ # - pragma — option_override — ( — func_name — [, " option — "] ) —▶▶
```

By default, optimization options specified on the command line apply to the entire C 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 great 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.

Allowable settings for *option* are:

|                    |                                                                   |
|--------------------|-------------------------------------------------------------------|
| opt (level, 2)     | Same as specifying <b>-O2</b> for the specified function.         |
| opt (level, 3)     | Same as specifying <b>-O3</b> for the specified function.         |
| opt (strict)       | Same as specifying <b>-qstrict</b> for the specified function.    |
| opt (strict, yes)  | Same as specifying <b>-qstrict</b> for the specified function.    |
| opt (strict, no)   | Same as specifying <b>-qnostrict</b> for the specified function.  |
| opt (compact)      | Same as specifying <b>-qcompact</b> for the specified function.   |
| opt (compact, yes) | Same as specifying <b>-qcompact</b> for the specified function.   |
| opt (compact, no)  | Same as specifying <b>-qnocompact</b> for the specified function. |

Selections for *option* are not subject to macro expansion.

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 Reading

“Preprocessor Directives” on page 58  
“#pragma Preprocessor Directives” on page 363  
“Compiler Options and Their Defaults” on page 218  
“List of Standard Preprocessor Directives” on page 347  
“O, optimize” on page 302  
“compact” on page 246  
“strict” on page 326

## #pragma options Preprocessor Directive

The **#pragma options** directives specifies compiler options within your source program.

```
▶▶ #pragma options (option) ▶▶
```

By default, the options specified apply to the entire C source program. If you specify more than one compiler option, use a blank space to separate them.

Most **#pragma options** directives must appear before any statements in your C program source. Comments and blank lines, however, may precede the **#pragma options** directive. For example, the first few lines of your C program can be a comment followed by the **#pragma options** directive, then the source:

```
/*  
 * The following is an example of a #pragma options directive:  
 */  
#pragma options langlvl=saa halt=s spill=1024 source  
/* The rest of the source follows below... */
```

For more information about compiler options, refer to “Compiler Options and Their Defaults” on page 218.

The following **#pragma options** directives can appear anywhere in the source file:

- **#pragma options source**
- **#pragma options enum**
- **#pragma options align**

Some **#pragma options** directives have corresponding preprocessor **#pragma** directives. These, along with their required placement locations in a C program source, are:

| #pragma options Name | #pragma Name  | Placement in Source                       |
|----------------------|---------------|-------------------------------------------|
| langlvl              | langlvl       | Before any statements in the source file. |
| chars                | chars         | Before any statements in the source file. |
| ro                   | strings       | Before any statements in the source file. |
| isolated_call        | isolated_call | Before any calls to the listed functions. |

**Note:** #pragma options arch=*suboption* is not supported in source files.

#### Related Reading

- “Preprocessor Directives” on page 58
- “#pragma Preprocessor Directives” on page 363
- “Compiler Options and Their Defaults” on page 218
- “List of Standard Preprocessor Directives” on page 347

## #pragma reachable Preprocessor Directive

The **#pragma reachable** directive takes a *function* name, and declares that the point in the program after that function can be the target of a branch from some unknown location. In other words, the instruction after the specified function can be reached from a program point other than the return statement in the named *function*.

```
▶▶ #pragma reachable ( 'function' , function ) ▶▶
```

If the specified function is not found, a warning message is shown.

#### Related Reading

- “Preprocessor Directives” on page 58
- “C Language Levels” on page 78
- “#pragma Preprocessor Directives” on page 363
- “List of Standard Preprocessor Directives” on page 347

## #pragma strings Preprocessor Directive

Specifies that the compiler can place strings into read-only memory, or must place strings into read/write memory.

```
▶▶ #pragma strings ( writable | readonly ) ▶▶
```

This pragma must appear before any statements in a source file. The default for **ansi** mode is **readonly**. The default for **extended** mode is **writable**. The specification **writable** is supported for portability between releases of the XL C compiler product.

This pragma has the same effect as the **-qro** compiler option.

### Related Reading

“Preprocessor Directives” on page 58

“#pragma Preprocessor Directives” on page 363

“List of Standard Preprocessor Directives” on page 347

“ro” on page 317

---

## Preprocessor Macro Operators

This page lists and briefly describes preprocessor macro operators available to you with the C for AIX compiler. To get more information on any item listed here, go to the reference page for that item.

| Preprocessor Macro Operators                   |                                                                                                |
|------------------------------------------------|------------------------------------------------------------------------------------------------|
| Name                                           | Action                                                                                         |
| “# Preprocessor Macro Operator”                | Converts a parameter of a function-like macro into a character string literal.                 |
| “## Preprocessor Macro Operator” on page 378   | Concatenates two tokens in a macro, ignoring white space between macro tokens and operators.   |
| “/**/ Preprocessor Macro Operator” on page 379 | Concatenates two tokens in a macro, preserving white space between macro tokens and operators. |

### Related Reading

“Preprocessor Directives” on page 58

“#define Preprocessor Directive” on page 348

“List of Standard Preprocessor Directives” on page 347

## # Preprocessor Macro Operator

The # (single number sign) operator converts a parameter of a function-like macro into a character string literal. For example, if macro ABC is defined using the following directive:

```
#define ABC(x) #x
```

all subsequent invocations of the macro ABC would be expanded into a character string literal containing the argument passed to ABC. For example:

| Invocation       | Result of Macro Expansion |
|------------------|---------------------------|
| ABC(1)           | "1"                       |
| ABC>Hello there) | "Hello there"             |

The # operator should not be confused with the “**# (Null) Preprocessor Directive” on page 347** null directive.

Use the # operator in a function-like macro definition according to the following rules:

- A parameter following # operator in a function-like macro is converted into a character string literal containing the argument passed to the macro.
- White-space characters that appear before or after the argument passed to the macro are deleted.
- Multiple white-space characters imbedded within the argument passed to the macro is replaced by a single space character.
- If the argument passed to the macro contains a string literal and if a \ (backslash) character appears within the literal, a second \ character is inserted before the original \ when the macro is expanded.

- If the argument passed to the macro contains a “ (double quotation mark) character, a \ character is inserted before the ” when the macro is expanded.
- If the argument passed to the macro contains a ’ (single quotation mark) character, a \ character is inserted before the ’ when the macro is expanded.
- The conversion of an argument into a string literal occurs before macro expansion on that argument.
- If more than one ## operator or # operator appears in the replacement list of a macro definition, the order of evaluation of the operators is not defined.
- If the result of the macro expansion is not a valid character string literal, the behavior is undefined.

The following example demonstrates the use of the # operator:

| Sample Preprocessor Macro Definitions                             |                           |
|-------------------------------------------------------------------|---------------------------|
| <pre>#define STR(x) #x #define XSTR(x) STR(x) #define ONE 1</pre> |                           |
| Invocation                                                        | Result of Macro Expansion |
| STR(\n "\n" '\n')                                                 | "\n \"\n\" '\n'"          |
| STR(ONE)                                                          | "ONE"                     |
| XSTR(ONE)                                                         | "1"                       |
| XSTR("hello")                                                     | "\hello"                  |

#### Related Reading

“Preprocessor Directives” on page 58

“Preprocessor Macros” on page 59

“Preprocessor Macro Operators” on page 377

“#define Preprocessor Directive” on page 348

## ## Preprocessor Macro Operator

Use the ## operator according to the following rules:

- The ## operator cannot be the very first or very last item in the replacement list of a macro definition.
- The last token of the item in front of the ## operator is concatenated with first token of the item following the ## operator.
- Concatenation takes place before any macros in arguments are expanded.
- If the result of a concatenation is a valid macro name, it is available for further replacement even if it appears in a context in which it would not normally be available.
- If more than one ## operator and/or “# Preprocessor Macro Operator” on page 377 operator appears in the replacement list of a macro definition, the order of evaluation of the operators is not defined.

The following examples demonstrate the use of the ## operator:

| Sample Preprocessor Macro Definitions                                                                                                                                     |                           |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------|
| <pre>#define ArgArg(x, y) x##y #define ArgText(x) x##TEXT #define TextArg(x) TEXT##x #define TextText TEXT##text #define Jitter 1 #define bug 2 #define Jitterbug 3</pre> |                           |
| Invocation                                                                                                                                                                | Result of Macro Expansion |
| ArgArg(lady, bug)                                                                                                                                                         | "ladybug"                 |

|                     |            |
|---------------------|------------|
| ArgText(con)        | "conTEXT"  |
| TextArg(book)       | "TEXTbook" |
| TextText            | "TEXTtext" |
| ArgArg(Jitter, bug) | 3          |

### Related Reading

- "Preprocessor Directives" on page 58
- "Preprocessor Macros" on page 59
- "Preprocessor Macro Operators" on page 377
- "#define Preprocessor Directive" on page 348

## /\*\*/ Preprocessor Macro Operator

The **/\*\*/** operator differs from the **## Preprocessor Macro Operator** on page 378 operator only in the way that the preprocessor treats white space between the operator and its arguments.

For example, the macro definition:

```
#define XY(x, y) x /**/y
```

does not give the same result as:

```
#define XY(x, y) x ##y
```

because the preprocessor preserves white space with the **/\*\*/** operator. With the **## Preprocessor Macro Operator** on page 378 operator, arguments are concatenated without white space.

The following examples demonstrate the use of the **/\*\*/** operator:

| Sample Preprocessor Macro Definitions                                                                           |                           |
|-----------------------------------------------------------------------------------------------------------------|---------------------------|
| <pre>#define ws1(x, y) x /**/y #define ws2(x, y) x/**/ y #define nws1(x, y) x ##y #define nws2(x, y) x##y</pre> |                           |
| Invocation                                                                                                      | Result of Macro Expansion |
| ws1(Turtle, neck)                                                                                               | Turtle neck               |
| ws2(Turtle, neck)                                                                                               | Turtle neck               |
| nws1(Turtle, neck)                                                                                              | Turtleneck                |
| nws2(Turtle, neck)                                                                                              | Turtleneck                |

For **/\*\*/** to function the same way as **## Preprocessor Macro Operator** on page 378 in ANSI/ISO C, there can be no spaces between the operator and the arguments.

### Related Reading

- "Preprocessor Directives" on page 58
- "Preprocessor Macros" on page 59
- "Preprocessor Macro Operators" on page 377
- "#define Preprocessor Directive" on page 348





## Appendix E. Parallel Processing Facilities

### #pragma Preprocessor Directives for 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 C for AIX-specific directives, and directives conforming to the OpenMP Application Program Interface specification.

All of the following pragmas have effect only if the **-qsmp** option is specified. If the **-qsmp** option is not specified, syntax checking of the pragmas is still performed even though corresponding code is not generated.

You can 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                                              | Description                                                                                                           |
|--------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| "#pragma ibm critical Preprocessor Directive" on page 382          | Instructs the compiler that the statement or statement block immediately following this pragma is a critical section. |
| "#pragma ibm independent_calls Preprocessor Directive" on page 383 | Asserts that specified function calls within the chosen loop have no loop-carried dependencies.                       |
| "#pragma ibm independent_loop Preprocessor Directive" on page 384  | Asserts that iterations of the chosen loop are independent, and that the loop can therefore be parallelized.          |
| "#pragma ibm iterations Preprocessor Directive" on page 384        | Specifies the approximate number of loop iterations for the chosen loop.                                              |
| "#pragma ibm parallel_loop Preprocessor Directive" on page 385     | Explicitly instructs the compiler to parallelize the chosen loop.                                                     |
| "#pragma ibm permutation Preprocessor Directive" on page 385       | Asserts that specified arrays in the chosen loop contain no repeated values.                                          |
| "#pragma ibm schedule Preprocessor Directive" on page 386          | Specifies scheduling algorithms for parallel loop execution.                                                          |
| "#pragma ibm sequential_loop Preprocessor Directive" on page 387   | Explicitly instructs the compiler to execute the chosen loop sequentially.                                            |

| OpenMP Pragma Directives                                      | Description                                                                                                                                                                            |
|---------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| "#pragma omp parallel Preprocessor Directive" on page 388     | Defines a parallel region to be run by multiple threads in parallel. With specific exceptions, all other OpenMP directives work within parallelized regions defined by this directive. |
| "#pragma omp for Preprocessor Directive" on page 389          | Work-sharing construct identifying an iterative for-loop whose iterations should be run in parallel.                                                                                   |
| "#pragma omp parallel for Preprocessor Directive" on page 393 | Shortcut combination of <b>omp parallel</b> and <b>omp for</b> pragma directives, used to define a parallel region containing a single <b>for</b> directive.                           |

|                                                                    |                                                                                                                                                                        |
|--------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| "#pragma omp sections Preprocessor Directive" on page 393          | 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 Preprocessor Directive" on page 394 | Shortcut combination of <b>omp parallel</b> and <b>omp sections</b> pragma directives, used to define a parallel region containing a single <b>sections</b> directive. |
| "#pragma omp single Preprocessor Directive" on page 395            | Work-sharing construct identifying a section of code that must be run by a single available thread.                                                                    |
| "#pragma omp master Preprocessor Directive" on page 395            | Synchronization construct identifying a section of code that must be run only by the master thread.                                                                    |
| "#pragma omp critical Preprocessor Directive" on page 396          | Synchronization construct identifying a statement block that must be executed by a single thread at a time.                                                            |
| "#pragma omp barrier Preprocessor Directive" on page 397           | Synchronizes all the threads in a parallel region.                                                                                                                     |
| "#pragma omp atomic Preprocessor Directive" on page 397            | Identifies a memory location that must be updated atomically and not be exposed to multiple, simultaneous writing threads.                                             |
| "#pragma omp flush Preprocessor Directive" on page 398             | 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 ordered Preprocessor Directive" on page 399           | Identifies a structure block of code that must be executed as a sequential loop.                                                                                       |
| "#pragma omp threadprivate Preprocessor Directive" on page 399     | Defines the scope of selected file-scope data variables as being private to a thread, but file-scope visible within that thread.                                       |

### Related Reading

- "Chapter 5. Program Parallelization" on page 37
- "Preprocessor Directives" on page 58
- "Using Pragmas to Control Parallel Processing" on page 41
- "List of Standard Preprocessor Directives" on page 347
- "Run-time Options for Parallel Processing" on page 402
- "OpenMP Run-time Options for Parallel Processing" on page 404
- "Built-in Functions Used for Parallel Processing" on page 400
- "smp" on page 320

## #pragma ibm critical Preprocessor Directive

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.

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 the **ibm critical** or **omp critical** directives are mapped to the same unspecified name.

## Related Reading

“Chapter 5. Program Parallelization” on page 37

“Shared and Private Variables in a Parallel Environment” on page 40

“Countable Loops” on page 38

“Using Pragmas to Control Parallel Processing” on page 41

“#pragma Preprocessor Directives for Parallel Processing” on page 381

“#pragma omp critical Preprocessor Directive” on page 396

“smp” on page 320

## #pragma ibm independent\_calls Preprocessor Directive

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.

## Example

```
/* #pragma ibm independent_calls */  
int s, a[100], i, N = 100;  
int foo (int);  
  
#pragma ibm independent_calls (foo)  
for (i = 0; i < N; i++) {  
    a[i] = foo(i);  
}
```

## Related Reading

“Chapter 5. Program Parallelization” on page 37

“Shared and Private Variables in a Parallel Environment” on page 40

“Countable Loops” on page 38

“Using Pragmas to Control Parallel Processing” on page 41

“#pragma Preprocessor Directives for Parallel Processing” on page 381

## #pragma ibm independent\_loop Preprocessor Directive

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. When the **if** argument is specified, loop iterations are considered independent only as long as *exp* evaluates to TRUE at run-time.

### Notes

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.

### Examples

```
/* #pragma ibm independent_loop applied to a for loop */  
#pragma ibm independent_loop  
for (i = 0; i < N; i++){  
    a[i] = i;  
}  
  
/* pragma independent_loop applied to a do-while loop */  
i = 0;  
#pragma ibm independent_loop  
do {  
    a[i] = i;  
    i++;  
} while (i < N);  
  
/* pragma independent_loop with if clause, applied to a while loop */  
i = 0;  
#pragma ibm independent_loop if (dist >= N/2)  
while (i < N/2) {  
    a[i] = a[i+dist];  
    i++;  
}
```

### Related Reading

- “Chapter 5. Program Parallelization” on page 37
- “Shared and Private Variables in a Parallel Environment” on page 40
- “Countable Loops” on page 38
- “Using Pragmas to Control Parallel Processing” on page 41
- “#pragma Preprocessor Directives for Parallel Processing” on page 381
- “#pragma ibm schedule Preprocessor Directive” on page 386

## #pragma ibm iterations Preprocessor Directive

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 Reading

“Chapter 5. Program Parallelization” on page 37  
“Shared and Private Variables in a Parallel Environment” on page 40  
“Countable Loops” on page 38  
“Using Pragmas to Control Parallel Processing” on page 41  
“#pragma Preprocessor Directives for Parallel Processing” on page 381

## #pragma ibm parallel\_loop Preprocessor Directive

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. 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.

### Notes

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 **ibm 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 **ibm schedule** pragma to select a specific parallel process scheduling algorithm. For more information, see the description for the **ibm schedule** pragma.

A warning is generated if this pragma is not followed by a countable loop.

### Example

```
/* #pragma ibm parallel_loop  
   The loop will execute in parallel if N is greater  
   or equal to 10000. Dynamic scheduling will be used. */  
#pragma ibm parallel_loop if (N >= 10000) schedule (dynamic)  
for (i = 0; i < N; i++) {  
    a[i] = z;  
}
```

### Related Reading

“Chapter 5. Program Parallelization” on page 37  
“Shared and Private Variables in a Parallel Environment” on page 40  
“Countable Loops” on page 38  
“Using Pragmas to Control Parallel Processing” on page 41  
“#pragma Preprocessor Directives for Parallel Processing” on page 381  
“#pragma ibm schedule Preprocessor Directive” on page 386“smp” on page 320

## #pragma ibm permutation Preprocessor Directive

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.

An array specified by this pragma cannot be a function parameter.

## Related Reading

“Chapter 5. Program Parallelization” on page 37

“Shared and Private Variables in a Parallel Environment” on page 40

“Countable Loops” on page 38

“Using Pragmas to Control Parallel Processing” on page 41

“#pragma Preprocessor Directives for Parallel Processing” on page 381

## #pragma ibm schedule Preprocessor Directive

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                                                                                                                                                                                                                     |
| affinity, <i>n</i> | <b>ceiling</b> ( <i>number_of_iterations/number_of_threads</i> ). Each local partition then further subdivided into chunks of size <b>ceiling</b> ( <i>number_of_iterations_remaining_in_partition/2</i> ).                                                                                  |
|                    | If <i>n</i> is specified, each local partition is subdivided into chunks of size <i>n</i> , where <i>n</i> is an integral assignment expression of value 1 or greater.                                                                                                                       |
|                    | 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.                                                                             |
| dynamic            | If <i>n</i> is not specified, iterations of a loop are divided into chunks of size 1.                                                                                                                                                                                                        |
| dynamic, <i>n</i>  | If <i>n</i> is specified, all chunks are set to size <i>n</i> , where <i>n</i> is an integral assignment expression of value 1 or greater.                                                                                                                                                   |
|                    | Chunks are assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.                                                                                                                                                   |
| guided             | Chunks are made progressively smaller until the default minimum chunk size is reached. The first chunk is of size <b>ceiling</b> ( <i>number_of_iterations/number_of_threads</i> ). Remaining chunks are of size <b>ceiling</b> ( <i>number_of_iterations_remaining/number_of_threads</i> ). |
| guided, <i>n</i>   | If <i>n</i> is specified, the minimum chunk size is set to <i>n</i> , where <i>n</i> is an integral assignment expression of value 1 or greater.                                                                                                                                             |
|                    | If <i>n</i> is not specified, a default value of 1 is assumed.                                                                                                                                                                                                                               |
|                    | Chunks are assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.                                                                                                                                                   |
| runtime            | Scheduling policy is determined at run-time.                                                                                                                                                                                                                                                 |

|                       |                                                                                                                                                                                                                                                                                                    |
|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>static</code>   | Iterations of a loop are divided into chunks of size <b>ceiling</b> ( <i>number_of_iterations/number_of_threads</i> ). Each thread is assigned a separate chunk.<br><br>This scheduling policy is also known as <i>block scheduling</i> .                                                          |
| <code>static,n</code> | 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.<br><br><i>n</i> must be an integral assignment expression of value 1 or greater.<br><br>This scheduling policy is also known as <i>block cyclic scheduling</i> . |
| <code>static,1</code> | Iterations of a loop are divided into chunks of size 1. Each chunk is assigned to a thread in <i>round-robin</i> fashion.<br><br>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** pragma statements. For example, the following sets of statements are equivalent:

```
#pragma ibm schedule (guided, 10)
#pragma ibm parallel_loop
for (i = 0; i < N; i++) {
    ...
}

and

#pragma ibm parallel_loop schedule (guided, 10)
for (i = 0; i < N; i++) {
    ...
}
```

If different scheduling types are specified for a given loop, the last one specified is applied.

## Related Reading

- “Chapter 5. Program Parallelization” on page 37
- “Shared and Private Variables in a Parallel Environment” on page 40
- “Countable Loops” on page 38
- “Using Pragmas to Control Parallel Processing” on page 41
- “#pragma Preprocessor Directives for Parallel Processing” on page 381
- “#pragma ibm parallel\_loop Preprocessor Directive” on page 385
- “Built-in Functions Used for Parallel Processing” on page 400
- “Run-time Options for Parallel Processing” on page 402
- “smp” on page 320

## #pragma ibm sequential\_loop Preprocessor Directive

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 Reading

“Chapter 5. Program Parallelization” on page 37

“Shared and Private Variables in a Parallel Environment” on page 40

“Countable Loops” on page 38

“Using Pragmas to Control Parallel Processing” on page 41

“#pragma Preprocessor Directives for Parallel Processing” on page 381

## #pragma omp parallel Preprocessor Directive

The **omp parallel** directive explicitly instructs the compiler to parallelize the chosen segment of code.

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.

## Syntax

```
#pragma omp parallel [clause[ clause] ...]  
<statement_block>
```

where *clause* is any of the following:

|                              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| if ( <i>exp</i> )            | When the <b>if</b> 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 <b>if</b> clause can be specified.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| 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 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.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| shared ( <i>list</i> )       | Declares the scope of the data variables in <i>list</i> to be shared across all threads. Data variables in <i>list</i> are separated by commas.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| copyin ( <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.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| default (shared   none)      | Each data variable specified in the <b>copyin</b> clause must be a <b>threadprivate</b> variable.<br>Defines the default data scope of variables in each thread. Only one <b>default</b> clause can be specified on an <b>omp parallel</b> directive.<br><br>Specifying <b>default(shared)</b> is equivalent to stating each variable in a <b>shared(list)</b> clause.<br><br>Specifying <b>default(none)</b> 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: <ul style="list-style-type: none"><li>• const-qualified,</li><li>• specified in an enclosed data scope attribute clause, or,</li><li>• used as a loop control variable referenced only by a corresponding <b>omp for</b> or <b>omp parallel for</b> directive.</li></ul> |



reduction (*operator*, *list*) Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas.

A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back into the original value of the shared reduction variable.

Variables specified in the **reduction** clause:

- must be of a type appropriate to the operator.
- must be shared in the enclosing context.
- must not be const-qualified.
- must not have pointer type.

## Notes

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 Reading

“Chapter 5. Program Parallelization” on page 37

“Shared and Private Variables in a Parallel Environment” on page 40

“Using Pragmas to Control Parallel Processing” on page 41

“#pragma Preprocessor Directives for Parallel Processing” on page 381

“OpenMP Run-time Options for Parallel Processing” on page 404

“#pragma omp parallel sections Preprocessor Directive” on page 394

## #pragma omp for Preprocessor Directive

The **omp for** directive instructs the compiler to distribute loop iterations within the team of threads that encounters this work-sharing construct.

## Syntax

```
#pragma omp for [clause [clause] ...]  
<for_loop>
```

where *clause* is any of the following:

|                              |                                                                                                                                                                                                                                                                                                                                                 |
|------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| private ( <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 iteration. Variables not assigned a value will have an indeterminate value. Data variables in <i>list</i> are separated by commas. |

|                                                |                                                                                                                                                                                                                                                                                                                              |
|------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| reduction ( <i>operator</i> :<br><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 <i>list</i> 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 <b>reduction</b> clause: <ul style="list-style-type: none"> <li>• must be of a type appropriate to the operator.</li> <li>• must be shared in the enclosing context.</li> <li>• must not be const-qualified.</li> <li>• must not have pointer type.</li> </ul>                                    |
| ordered                                        | Specify this clause if an ordered construct is present within the dynamic extent of the <b>omp for</b> directive.                                                                                                                                                                                                            |

schedule (*type*)

Specifies how iterations of the **for** loop are divided among available threads. Acceptable values for *type* are:

dynamic  
dynamic,*n*

If *n* is not specified, iterations of a loop are divided into chunks of size **ceiling**(*number\_of\_iterations/number\_of\_threads*).

If *n* is specified, all chunks are set to size *n*. *n* must be an integral assignment expression of value 1 or greater.

Chunks are dynamically assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.

guided  
guided,*n*

Chunks are made progressively smaller until the default minimum chunk size is reached. The first chunk is of size **ceiling**(*number\_of\_iterations/number\_of\_threads*). Remaining chunks are of size **ceiling**(*number\_of\_iterations\_remaining/number\_of\_threads*).

If *n* is specified, the minimum chunk size is set to *n*. *n* must be an integral assignment expression of value 1 or greater.

If *n* is not specified, a default value of 1 is assumed.

Chunks are assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.

runtime

Scheduling policy is determined at run-time. Use the OMP\_SCHEDULE environment variable to set the scheduling type and chunk size.

static

Iterations of a loop are divided into chunks of size **ceiling**(*number\_of\_iterations/number\_of\_threads*). Each thread is assigned a separate chunk.

This scheduling policy is also known as *block scheduling*.

static,*n*

Iterations of a loop are divided into chunks of size *n*. Each chunk is assigned to a thread in *round-robin* fashion.

*n* must be an integral assignment expression of value 1 or greater.

This scheduling policy is also known as *block cyclic scheduling*.

static,1

Iterations of a loop are divided into chunks of size 1. Each chunk is assigned to a thread in *round-robin* fashion.

This scheduling policy is also known as *cyclic scheduling*.

`nowait` Use this clause to avoid the implied **barrier** at the end of the **for** directive. This is useful if you have multiple independent work-sharing sections or iterative loops within a given parallel region. Only one **nowait** clause can appear on a given **for** directive.

and where *for\_loop* is a **for** loop construct with the following canonical shape:

```
for (init_expr; exit_cond; incr_expr)
    statement
```

where:

|                  |             |                                                                                                                                                                                                                                           |
|------------------|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>init_expr</i> | takes form: | <i>iv</i> = <i>b</i><br><i>integer-type iv</i> = <i>b</i>                                                                                                                                                                                 |
| <i>exit_cond</i> | takes form: | <i>iv</i> <= <i>ub</i><br><i>iv</i> < <i>ub</i><br><i>iv</i> >= <i>ub</i><br><i>iv</i> > <i>ub</i>                                                                                                                                        |
| <i>incr_expr</i> | takes form: | ++ <i>iv</i><br><i>iv</i> ++<br>-- <i>iv</i><br><i>iv</i> --<br><i>iv</i> += <i>incr</i><br><i>iv</i> -= <i>incr</i><br><i>iv</i> = <i>iv</i> + <i>incr</i><br><i>iv</i> = <i>incr</i> + <i>iv</i><br><i>iv</i> = <i>iv</i> - <i>incr</i> |

and:

|                                    |                                                                                                                                                                                                                                                                                                                                    |
|------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>iv</i>                          | Iteration variable. The iteration variable must be a signed integer not modified anywhere within the <b>for</b> loop. It is implicitly made private for the duration of the <b>for</b> operation. If not specified as <b>lastprivate</b> , the iteration variable will have an indeterminate value after the operation completes.. |
| <i>b</i> , <i>ub</i> , <i>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 Reading

“Chapter 5. Program Parallelization” on page 37  
“Shared and Private Variables in a Parallel Environment” on page 40  
“Using Pragmas to Control Parallel Processing” on page 41  
“#pragma Preprocessor Directives for Parallel Processing” on page 381  
“#pragma omp ordered Preprocessor Directive” on page 399  
“OpenMP Run-time Options for Parallel Processing” on page 404

## #pragma omp parallel for Preprocessor Directive

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 **omp for** directive in one step.

### Syntax

```
#pragma omp parallel for [clause [clause] ...]  
<for_loop>
```

All clauses and restrictions described in the **omp parallel** and **omp for** directives apply to the **omp parallel for** directive.

## Related Reading

“Chapter 5. Program Parallelization” on page 37  
“Shared and Private Variables in a Parallel Environment” on page 40  
“Countable Loops” on page 38  
“Using Pragmas to Control Parallel Processing” on page 41  
“#pragma Preprocessor Directives for Parallel Processing” on page 381  
“OpenMP Run-time Options for Parallel Processing” on page 404  
“#pragma omp parallel Preprocessor Directive” on page 388  
“#pragma omp for Preprocessor Directive” on page 389

## #pragma omp sections Preprocessor Directive

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 <b>section</b> . Variables not assigned a value will have an indeterminate value. Data variables in <i>list</i> are separated by commas. |

|                                                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| reduction ( <i>operator</i> :<br><i>list</i> ) | <p>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.</p> <p>A private copy of each variable in <i>list</i> 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.</p> <p>Variables specified in the <b>reduction</b> clause:</p> <ul style="list-style-type: none"> <li>• must be of a type appropriate to the operator.</li> <li>• must be shared in the enclosing context.</li> <li>• must not be const-qualified.</li> <li>• must not have pointer type.</li> </ul> |
| nowait                                         | <p>Use this clause to avoid the implied <b>barrier</b> at the end of the <b>sections</b> directive. This is useful if you have multiple independent work-sharing sections within a given parallel region. Only one <b>nowait</b> clause can appear on a given <b>sections</b> directive.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |

## 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 Reading

“Chapter 5. Program Parallelization” on page 37  
 “Shared and Private Variables in a Parallel Environment” on page 40  
 “Using Pragmas to Control Parallel Processing” on page 41  
 “#pragma Preprocessor Directives for Parallel Processing” on page 381  
 “OpenMP Run-time Options for Parallel Processing” on page 404  
 “#pragma omp parallel sections Preprocessor Directive”

## #pragma omp parallel sections Preprocessor Directive

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 **omp sections** directive in one step.

## Syntax

```
#pragma omp parallel sections [clause[ clause] ...]
{
    [#pragma omp section]
    statement-block
    [#pragma omp section]
    statement-block
    .
    .
    .
}
```

All clauses and restrictions described in the **omp parallel** and **omp sections** directives apply to the **omp parallel sections** directive.

### Related Reading

“Chapter 5. Program Parallelization” on page 37  
“Shared and Private Variables in a Parallel Environment” on page 40  
“Countable Loops” on page 38  
“Using Pragmas to Control Parallel Processing” on page 41  
“#pragma Preprocessor Directives for Parallel Processing” on page 381  
“OpenMP Run-time Options for Parallel Processing” on page 404  
“#pragma omp parallel Preprocessor Directive” on page 388  
“#pragma omp sections Preprocessor Directive” on page 393

## #pragma omp single Preprocessor Directive

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 <b>barrier</b> at the end of the <b>single</b> directive. Only one <b>nowait</b> clause can appear on a given <b>single</b> directive.                                                                            |

### Notes

An implied barrier exists at the end of a parallelized statement block unless the **nowait** clause is specified.

### Related Reading

“Chapter 5. Program Parallelization” on page 37  
“Shared and Private Variables in a Parallel Environment” on page 40  
“Using Pragmas to Control Parallel Processing” on page 41  
“#pragma Preprocessor Directives for Parallel Processing” on page 381  
“OpenMP Run-time Options for Parallel Processing” on page 404

## #pragma omp master Preprocessor Directive

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 Reading

“Chapter 5. Program Parallelization” on page 37  
“Shared and Private Variables in a Parallel Environment” on page 40  
“Countable Loops” on page 38  
“Using Pragmas to Control Parallel Processing” on page 41  
“#pragma Preprocessor Directives for Parallel Processing” on page 381  
“OpenMP Run-time Options for Parallel Processing” on page 404

## #pragma omp critical Preprocessor Directive

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 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.

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 the **ibm critical** or **omp critical** directives are mapped to the same unspecified name.

### Example

```
/* #pragma ibm critical */  
int s, a[100], i;  
  
#pragma ibm parallel_loop  
for (i = 0; i < 100; i++) {  
    #pragma ibm critical (lock1)  
    s = s + a[i]*i;  
}
```

### Related Reading

“Chapter 5. Program Parallelization” on page 37  
“Shared and Private Variables in a Parallel Environment” on page 40  
“Countable Loops” on page 38  
“Using Pragmas to Control Parallel Processing” on page 41  
“#pragma Preprocessor Directives for Parallel Processing” on page 381  
“#pragma ibm critical Preprocessor Directive” on page 382  
“OpenMP Run-time Options for Parallel Processing” on page 404



## #pragma omp barrier Preprocessor Directive

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 Reading

“Chapter 5. Program Parallelization” on page 37

“Shared and Private Variables in a Parallel Environment” on page 40

“Countable Loops” on page 38

“Using Pragmas to Control Parallel Processing” on page 41

“#pragma Preprocessor Directives for Parallel Processing” on page 381

“OpenMP Run-time Options for Parallel Processing” on page 404

## #pragma omp atomic Preprocessor Directive

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
```

where *statement* is an expression statement of scalar type that takes one of the forms that follow:

```
x bin_op = expr
```

where:

```
bin_op
```

is one of:

```
+ * - / & ^ | << >>
```

*expr* is an expression of scalar type that does not reference *x*.

```
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 Reading

“Chapter 5. Program Parallelization” on page 37  
“Shared and Private Variables in a Parallel Environment” on page 40  
“Countable Loops” on page 38  
“Using Pragmas to Control Parallel Processing” on page 41  
“#pragma Preprocessor Directives for Parallel Processing” on page 381  
“OpenMP Run-time Options for Parallel Processing” on page 404

## #pragma omp flush Preprocessor Directive

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 **flush** directive is implied 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 Reading

- “Chapter 5. Program Parallelization” on page 37
- “Shared and Private Variables in a Parallel Environment” on page 40
- “Countable Loops” on page 38
- “Using Pragmas to Control Parallel Processing” on page 41
- “#pragma Preprocessor Directives for Parallel Processing” on page 381
- “OpenMP Run-time Options for Parallel Processing” on page 404

## #pragma omp ordered Preprocessor Directive

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 Reading

- “Chapter 5. Program Parallelization” on page 37
- “Shared and Private Variables in a Parallel Environment” on page 40
- “Countable Loops” on page 38
- “Using Pragmas to Control Parallel Processing” on page 41
- “#pragma Preprocessor Directives for Parallel Processing” on page 381
- “#pragma omp for Preprocessor Directive” on page 389
- “OpenMP Run-time Options for Parallel Processing” on page 404

## #pragma omp threadprivate Preprocessor Directive

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 Reading

“Chapter 5. Program Parallelization” on page 37

“Shared and Private Variables in a Parallel Environment” on page 40

“Countable Loops” on page 38

“Using Pragmas to Control Parallel Processing” on page 41

“#pragma Preprocessor Directives for Parallel Processing” on page 381

“OpenMP Run-time Options for Parallel Processing” on page 404

---

## Built-in Functions Used for Parallel Processing

Your program can use these built-in functions to modify or 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>                              | <p>This function returns the value of the <b>parthds</b> run-time option. If the <b>parthds</b> option is not explicitly set by the user, the function returns the default value set by the run-time library.</p> <p>If the <b>-qsmp</b> compiler option was not specified during program compilation, this function returns 1 regardless of run-time options selected.</p>                                                                                                                                                                                                                                                                                       |
| <code>int __usrthds(void)</code>                              | <p>This function returns the value of the <b>usrthds</b> run-time option.</p> <p>If the <b>usrthds</b> option is not explicitly set by the user, or the <b>-qsmp</b> compiler option was not specified during program compilation, this function returns 0 regardless of run-time options selected.</p>                                                                                                                                                                                                                                                                                                                                                           |
| <code>int omp_set_num_threads(int <i>num_threads</i>);</code> | <p>This function must be called from a serial portion of your program.</p> <ul style="list-style-type: none"> <li>• If dynamic adjustment of the number of threads is enabled, this function sets <i>num_threads</i> as the maximum number of threads to use for subsequent parallel regions.</li> <li>• If dynamic adjustment of the number of threads is disabled, this function causes <i>num_threads</i> to be used as the number of threads for all subsequent parallel regions.</li> </ul> <p>The value of <i>num_threads</i> must be a positive integer. This function takes overrides the setting of the <b>OMP_NUM_THREADS</b> environment variable.</p> |

| Function Prototype                                                                                                       | Description                                                                                                                                                                                                                                                                                 |
|--------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <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.                                                                                                                                                                                                                                       |
| <code>int omp_get_nested(void);</code>                                                                                   | This function returns non-zero if nested parallelism is enabled and 0 if it is disabled.                                                                                                                                                                                                    |
| <code>void omp_init_lock(omp_lock_t *lock);</code><br><code>void omp_init_nest_lock(omp_nest_lock_t *lock);</code>       | These functions provide the only means of initializing a lock. Each function initializes the lock associated with the parameter <code>lock</code> for use in subsequent calls.                                                                                                              |
| <code>void omp_destroy_lock(omp_lock_t *lock);</code><br><code>void omp_destroy_nest_lock(omp_nest_lock_t *lock);</code> | These functions ensure that the pointed to lock variable <code>lock</code> is uninitialized.                                                                                                                                                                                                |
| <code>void omp_set_lock(omp_lock_t *lock);</code><br><code>void omp_set_nest_lock(omp_nest_lock_t *lock);</code>         | 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. |
| <code>void omp_unset_lock(omp_lock_t *lock);</code><br><code>void omp_unset_nest_lock(omp_nest_lock_t *lock);</code>     | These functions provide the means of releasing ownership of a lock.                                                                                                                                                                                                                         |
| <code>int omp_test_lock(omp_lock_t *lock);</code><br><code>int omp_test_nest_lock(omp_nest_lock_t *lock);</code>         | 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 Reading

“Chapter 5. Program Parallelization” on page 37  
“Shared and Private Variables in a Parallel Environment” on page 40  
“Countable Loops” on page 38  
“#pragma Preprocessor Directives for Parallel Processing” on page 381  
“Run-time Options for Parallel Processing”  
“OpenMP Run-time Options for Parallel Processing” on page 404  
“smp” on page 320

---

## Run-time Options for Parallel Processing

Run-time options affecting parallel processing are specified in the XLSMPOPTS environment variable. This environment variable, which must be set before you run an application, uses syntax of form:

```
XLSMPOPTS=option_and_args[:option_and_args][ ... ]
```

Parallelization run-time options can also be specified using OMP environment variables. When OMP and XLSMPOPTS run-time options conflict, OMP options will prevail.

Run-time options fall into different categories as described below.

### Scheduling Algorithm Options

`schedule=algorithm=[n]` 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 Preprocessor Directive” on page 386 for a description of these algorithms.

### Parallel Environment Options

`parthds=num`

*num* represents the number of parallel threads requested, which is usually equivalent to the number of processors available on the system.

Some applications cannot use more threads than the maximum number of processors available. Other applications can experience significant performance improvements if they use more threads than there are processors. This option gives you full control over the number of user threads used to run your program.

The default value for *num* is the number of processors available on the system. *num* represents the number of user threads expected.

`usrthds=num`

This option should be used if the program code explicitly creates threads, in which case *num* should be set to the number of threads created.

The default value for *num* is 0.

`stack=num` *num* specifies the largest amount of space required for a thread's stack.  
The default value for *num* is 32768.

## Performance Tuning Options

`spins=num` *num* represents the number of loop spins before a yield occurs.  
When a thread completes its work, the thread continues executing in a tight loop looking for new work. One complete scan of the work queue is done during each busy-wait state. An extended busy-wait state can make a particular application highly responsive, but can also harm the overall responsiveness of the system unless the thread is given instructions to periodically scan for and yield to requests from other applications.

A complete busy-wait state for benchmarking purposes can be forced by setting both **spins** and **yields** to 0.

The default value for *num* is 100.

`yields=num` *num* represents the number of yields before a sleep occurs.

When a thread sleeps, it completely suspends execution until another thread signals that there is work to do. This provides better system utilization, but also adds extra system overhead for the application.

The default value for *num* is 100.

`delays=num` *num* represents a period of do-nothing delay time between each scan of the work queue. Each unit of delay is achieved by running a single no-memory-access delay loop.

The default value for *num* is 500.

## Dynamic Profiling Options

`profilefreq=num` *num* represents the sampling rate at which each loop is revisited to determine appropriateness for parallel processing.

The run-time library uses dynamic profiling to dynamically tune the performance of automatically-parallelized loops. Dynamic profiling gathers information about loop running times to determine if the loop should be run sequentially or in parallel the next time through. Threshold running times are set by the **parthreshold** and **seqthreshold** dynamic profiling options, described below.

If *num* is 0, all profiling is turned off, and overheads that occur because of profiling will not occur. If *num* is greater than 0, running time of the loop is monitored once every *num* times through the loop.

The default for *num* is 16. The maximum sampling rate is 32. Higher values of *num* are changed to 32.

`parthreshold=mSec` *mSec* specifies the expected running time in milliseconds below which a loop must be run sequentially. *mSec* can be specified using decimal places.

If **parthreshold** is set to 0, a parallelized loop will never be serialized by the dynamic profiler.

The default value for *mSec* is 0.2 milliseconds.

`seqthreshold=mSec` *mSec* specifies the expected running time in milliseconds beyond which a loop that has been serialized by the dynamic profiler must revert to being run in parallel mode again. *mSec* can be specified using decimal places.

The default value for *mSec* is 5 milliseconds.

**Note:**

You must use thread-safe compiler mode invocations when compiling parallelized program code.

**Related Reading**

“Chapter 5. Program Parallelization” on page 37  
 “Shared and Private Variables in a Parallel Environment” on page 40  
 “Countable Loops” on page 38  
 “Compiler Modes” on page 5  
 “Invoking the Compiler” on page 8  
 “#pragma Preprocessor Directives for Parallel Processing” on page 381  
 “#pragma Preprocessor Directives for Parallel Processing” on page 381  
 “OpenMP Run-time Options for Parallel Processing”  
 “#pragma ibm schedule Preprocessor Directive” on page 386  
 “Built-in Functions Used for Parallel Processing” on page 400  
 “smp” on page 320

---

## OpenMP Run-time Options for Parallel Processing

OpenMP run-time options affecting parallel processing are specified in a set of OMP environment variables. These environment variables, which must be set before you run an application, use syntax of 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.

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 is scheduling algorithm is **static**.

See “#pragma omp for Preprocessor Directive” on page 389 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.

`OMP_NESTED=TRUE|FALSE` The default value for *num* is the number of processors available on the system. 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 “Run-time Options for Parallel Processing” on page 402 for more information.

If set to `FALSE`, dynamic adjustment is disabled.

The default setting is `TRUE`.

### Note:

You must use thread-safe compiler mode invocations when compiling parallelized program code.

### Related Reading

“Chapter 5. Program Parallelization” on page 37  
“Shared and Private Variables in a Parallel Environment” on page 40  
“Compiler Modes” on page 5  
“Invoking the Compiler” on page 8  
“#pragma Preprocessor Directives for Parallel Processing” on page 381  
“Run-time Options for Parallel Processing” on page 402  
“#pragma ibm schedule Preprocessor Directive” on page 386  
“Built-in Functions Used for Parallel Processing” on page 400  
“smp” on page 320



---

## Appendix F. C for AIX Debug Functions

---

### **`_debug_calloc` - Allocate and Initialize Memory**

#### **Format**

```
#include <stdlib.h> /* also in <malloc.h> */
void *_debug_calloc(size_t num, size_t size,
                   const char *file, size_t line);
```

**Language Level:** Extension

`_debug_calloc` is the debug version of `calloc`. Like `calloc`, it allocates memory from the default heap for an array of *num* elements, each of length *size* bytes. It then initializes all bits of each element to 0.

In addition, `_debug_calloc` makes an implicit call to `_heap_check`, and stores the name of the file *file* and the line number *line* where the storage is allocated. This information can be used later by the `_heap_check`, `_dump_allocated` or `_dump_allocated_delta` functions.

To use `_debug_calloc`, you must compile with the debug memory `-qheapdebug` compiler option. This option maps all `calloc` calls to `_debug_calloc`.

**Note:** The `-qheapdebug` option maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

To reallocate or free memory allocated by `_debug_calloc`, use `_debug_realloc` and `_debug_free`; you can also use `realloc` and `free` if you do not want debug information about the operation.

A heap-specific version (`_debug_ucalloc`) is available. `_debug_calloc` always allocates memory from the default heap.

#### **Return Value**

`_debug_calloc` returns a pointer to the reserved space. If not enough memory is available, or if *num* or *size* is 0, `_debug_calloc` returns NULL.

#### **Example**

This example reserves storage of 100 bytes. It then attempts to write to storage that was not allocated. When `_debug_calloc` is called again, `_heap_check` detects the error, generates several messages, and stops the program.

**Note:** You must compile this example with the `-qheapdebug` option to map the `calloc` calls to `_debug_calloc`.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *ptr1, *ptr2;
    if (NULL == (ptr1 = (char*)calloc(1, 100))) {
        puts("Could not allocate memory block.");
        exit(EXIT_FAILURE);
    }
    memset(ptr1, 'a', 105); /* overwrites storage that was not allocated */
    ptr2 = (char*)calloc(2, 20); /* this call to calloc invokes _heap_check */
    puts("_debug_calloc did not detect that a memory block was overwritten.");
    return 0;
}
```

```

/*****
The output should be similar to:
End of allocated object 0x00073890 was overwritten at 0x000738f4.
The first eight bytes of the memory block (in hex) are: 6161616161616161.
This memory block was (re)allocated at line number 9 in _debug_callo.c.
Heap state was valid at line 9 of _debug_callo.c.
Memory error detected at line 14 of _debug_callo.c.
*****/
}

```

## Related Reading

- “Debugging Memory Heaps” on page 183
- “Memory Management Functions” on page 179
- “Managing Memory with Multiple Heaps” on page 182
- “Debugging Programs with Heap Memory” on page 195
- “heapdebug” on page 270
- “\_debug\_free - Free Allocated Memory”
- “\_debug\_heapmin - Free Unused Memory in the Default Heap” on page 410
- “\_debug\_malloc - Allocate Memory” on page 412
- “\_debug\_memcpy - Copy Bytes” on page 413
- “\_debug\_memmove - Copy Bytes” on page 415
- “\_debug\_memset - Set Bytes to Value” on page 416
- “\_debug\_realloc - Reallocate Memory Block” on page 417
- “\_debug\_strcat - Concatenate Strings” on page 419
- “\_debug\_strcpy - Copy Strings” on page 421
- “\_debug\_strncat - Concatenate Strings” on page 422
- “\_debug\_strncpy - Copy Strings” on page 423
- “\_debug\_strnset - Set Characters in String” on page 425
- “\_debug\_strset - Set Characters in String” on page 426
- “\_debug\_ucallo - Reserve and Initialize Memory from User Heap” on page 428
- “\_debug\_uheapmin - Free Unused Memory in User Heap” on page 430
- “\_debug\_umalloc - Reserve Memory Blocks from User Heap” on page 431
- “heapdebug” on page 270

---

## **`_debug_free` - Free Allocated Memory**

### Format

```

#include <stdlib.h> /* also in <malloc.h> */
void _debug_free(void *ptr, const char *file,
                 size_t line);

```

### Language Level: Extension

`_debug_free` is the debug version of `free`. Like `free`, it frees the block of memory pointed to by `ptr`. `_debug_free` also sets each block of freed memory to 0xFB, so you can easily locate instances where your program uses the data in freed memory.

In addition, `_debug_free` makes an implicit call to the `_heap_check`, and stores the file name `file` and the line number `line` where the memory is freed. This information can be used later by the `_heap_check`, `_dump_allocated`, or `_dump_allocated_delta` functions.

To use `_debug_free`, you must compile with the debug memory `-qheapdebug` compiler option. This option maps all free calls to `_debug_free`.

**Note:**The `-qheapdebug` option maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Because `_debug_free` always checks what heap the memory was allocated from, you can use `_debug_free` to free memory blocks allocated by the regular, heap-specific, or debug versions of the memory management functions. However, if the memory was not allocated by the memory management functions, or was previously freed, `_debug_free` generates an error message and the program ends.

### Return Value

There is no return value.

### Example

This example reserves two blocks, one of 10 bytes and the other of 20 bytes. It then frees the first block and attempts to overwrite the freed storage. When `_debug_free` is called a second time, `_heap_check` detects the error, prints out several messages, and stops the program.

**Note:** You must compile this example with the `-qheapdebug` option to map the free calls to `_debug_free`.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *ptr1, *ptr2;
    if (NULL == (ptr1 = (char*)malloc(10)) || NULL == (ptr2 = (char*)malloc(20))) {
        puts("Could not allocate memory block.");
        exit(EXIT_FAILURE);
    }
    free(ptr1);
    memset(ptr1, 'a', 5);      /* overwrites storage that has been freed */
    free(ptr2);               /* this call to free invokes _heap_check */
    puts("_debug_free did not detect that a freed memory block was overwritten.");
    return 0;

    /*****
     The output should be similar to:
     Free heap was overwritten at 0x00073890.
     Heap state was valid at line 12 of _debug_free.c.
     Memory error detected at line 14 of _debug_free.c.
     *****/
}
```

## Related Reading

“Debugging Memory Heaps” on page 183  
“Memory Management Functions” on page 179  
“Managing Memory with Multiple Heaps” on page 182  
“Debugging Programs with Heap Memory” on page 195  
“heapdebug” on page 270  
“\_debug\_calloc - Allocate and Initialize Memory” on page 407  
“\_debug\_heapmin - Free Unused Memory in the Default Heap”  
“\_debug\_malloc - Allocate Memory” on page 412  
“\_debug\_memcpy - Copy Bytes” on page 413  
“\_debug\_memmove - Copy Bytes” on page 415  
“\_debug\_memset - Set Bytes to Value” on page 416  
“\_debug\_realloc - Reallocate Memory Block” on page 417  
“\_debug\_strcat - Concatenate Strings” on page 419  
“\_debug\_strcpy - Copy Strings” on page 421  
“\_debug\_strncat - Concatenate Strings” on page 422  
“\_debug\_strncpy - Copy Strings” on page 423  
“\_debug\_strnset - Set Characters in String” on page 425  
“\_debug\_strset - Set Characters in String” on page 426  
“\_debug\_ucalloc - Reserve and Initialize Memory from User Heap” on page 428  
“\_debug\_uheapmin - Free Unused Memory in User Heap” on page 430  
“\_debug\_umalloc - Reserve Memory Blocks from User Heap” on page 431  
“heapdebug” on page 270

---

## **\_debug\_heapmin - Free Unused Memory in the Default Heap**

### Format

```
#include <stdlib.h> /* also in <malloc.h> */
int _debug_heapmin(const char *file, size_t line);
```

**Language Level:** Extension

**\_debug\_heapmin** is the debug version of **\_heapmin**. Like **\_heapmin**, it returns all unused memory from the default runtime heap to the operating system.

In addition, **\_debug\_heapmin** makes an implicit call to **\_heap\_check**, and stores the file name *file* and the line number *line* where the memory is returned. This information can be used later by the **\_heap\_check** function.

To use **\_debug\_heapmin**, you must compile with the debug memory **-qheapdebug** compiler option. This option maps all **\_heapmin** calls to **\_debug\_heapmin**.

**Note:**The **-qheapdebug** option maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

A heap-specific version of this function (**\_debug\_uheapmin**) is also available. **\_debug\_heapmin** always operates on the default heap.

### Return Value

If successful, **\_debug\_heapmin** returns 0; otherwise, it returns -1.

### Example

This example allocates 10000 bytes of storage, changes the storage size to 10 bytes, and then uses **\_debug\_heapmin** to return the unused memory to the operating system. The program then attempts to

overwrite memory that was not allocated. When `_debug_heapmin` is called again, `_heap_check` detects the error, generates several messages, and stops the program.

**Note:** You must compile this example with the `-qheapdebug` option to map the `_heapmin` calls to `_debug_heapmin`.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *ptr;

    /* Allocate a large object from the system */
    if (NULL == (ptr = (char*)malloc(100000))) {
        puts("Could not allocate memory block.");
        exit(EXIT_FAILURE);
    }
    ptr = (char*)realloc(ptr, 10);
    _heapmin(); /* No allocation problems to detect */
    *(ptr - 1) = 'a'; /* Overwrite memory that was not allocated */
    _heapmin(); /* This call to _heapmin invokes _heap_check */
    puts("_debug_heapmin did not detect that a non-allocated memory block"
        "was overwritten.");
    return 0;

    /*****
    Possible output is:

    Header information of object 0x000738b0 was overwritten at 0x000738ac.
    The first eight bytes of the memory block (in hex) are: AAAAAAAAAAAAAA.
    This memory block was (re)allocated at line number 13 in _debug_heapm.c.
    Heap state was valid at line 14 of _debug_heapm.c.
    Memory error detected at line 17 of _debug_heapm.c.
    *****/
}
```

### Related Reading

- "Debugging Memory Heaps" on page 183
- "Memory Management Functions" on page 179
- "Managing Memory with Multiple Heaps" on page 182
- "Debugging Programs with Heap Memory" on page 195
- "heapdebug" on page 270
- "\_debug\_calloc - Allocate and Initialize Memory" on page 407
- "\_debug\_free - Free Allocated Memory" on page 408
- "\_debug\_malloc - Allocate Memory" on page 412
- "\_debug\_memcpy - Copy Bytes" on page 413
- "\_debug\_memmove - Copy Bytes" on page 415
- "\_debug\_memset - Set Bytes to Value" on page 416
- "\_debug\_realloc - Reallocate Memory Block" on page 417
- "\_debug\_strcat - Concatenate Strings" on page 419
- "\_debug\_strcpy - Copy Strings" on page 421
- "\_debug\_strncat - Concatenate Strings" on page 422
- "\_debug\_strncpy - Copy Strings" on page 423
- "\_debug\_strnset - Set Characters in String" on page 425
- "\_debug\_strset - Set Characters in String" on page 426
- "\_debug\_ucalloc - Reserve and Initialize Memory from User Heap" on page 428
- "\_debug\_uheapmin - Free Unused Memory in User Heap" on page 430
- "\_debug\_umalloc - Reserve Memory Blocks from User Heap" on page 431
- "heapdebug" on page 270

---

## **\_\_debug\_malloc - Allocate Memory**

### **Format**

```
#include <stdlib.h> /* also in <malloc.h> */
void *__debug_malloc(size_t size,
                    const char *file, size_t line);
```

### **Language Level:** Extension

**\_\_debug\_malloc** is the debug version of **malloc**. Like **malloc**, it reserves a block of storage of *size* bytes from the default heap. **\_\_debug\_malloc** also sets all the memory it allocates to 0xAA, so you can easily locate instances where your program uses the data in the memory without initializing it first.

In addition, **\_\_debug\_malloc** makes an implicit call to **\_\_heap\_check**, and stores the file name *file* and the line number *line* where the storage is allocated. This information can later be used by the **\_\_heap\_check**, **\_\_dump\_allocated**, or **\_\_dump\_allocated\_delta** functions.

To use **\_\_debug\_malloc**, you must compile with the debug memory **-qheapdebug** compiler option. This option maps all **malloc** calls to **\_\_debug\_malloc**.

**Note:** The **-qheapdebug** option maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

To reallocate or free memory allocated by **\_\_debug\_malloc**, use **\_\_debug\_realloc** and **\_\_debug\_free**; you can also use **realloc** and **free** if you do not want debug information about the operation.

A heap-specific version of this function (**\_\_debug\_umalloc**) is also available. **\_\_debug\_malloc** always allocates memory from the default heap.

### **Return Value**

**\_\_debug\_malloc** returns a pointer to the reserved space. If not enough memory is available or if *size* is 0, **\_\_debug\_malloc** returns NULL.

### **Example**

This example allocates 100 bytes of storage. It then attempts to write to storage that was not allocated. When **\_\_debug\_malloc** is called again, **\_\_heap\_check** detects the error, generates several messages, and stops the program.

**Note:** You must compile this example with the **-qheapdebug** option to map the **malloc** calls to **\_\_debug\_malloc**.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *ptr1, *ptr2;
    if (NULL == (ptr1 = (char*)malloc(100))) {
        puts("Could not allocate memory block.");
        exit(EXIT_FAILURE);
    }
    *(ptr1 - 1) = 'a'; /* overwrites storage that was not allocated */
    ptr2 = (char*)malloc(10); /* this call to malloc invokes __heap_check */
    puts("__debug_malloc did not detect that a memory block was overwritten.");
    return 0;
}

/*****
Possible output is:
```



```

Header information of object 0x00073890 was overwritten at 0x0007388c.
The first eight bytes of the memory block (in hex) are: AAAAAAAAAAAAAAAA.
This memory block was (re)allocated at line number 8 in _debug_mallo.c.
Heap state was valid at line 8 of _debug_mallo.c.
Memory error detected at line 13 of _debug_mallo.c.
*****/
}

```

### Related Reading

- “Debugging Memory Heaps” on page 183
- “Memory Management Functions” on page 179
- “Managing Memory with Multiple Heaps” on page 182
- “Debugging Programs with Heap Memory” on page 195
- “heapdebug” on page 270
- “\_debug\_calloc - Allocate and Initialize Memory” on page 407
- “\_debug\_free - Free Allocated Memory” on page 408
- “\_debug\_heapmin - Free Unused Memory in the Default Heap” on page 410
- “\_debug\_memcpy - Copy Bytes”
- “\_debug\_memmove - Copy Bytes” on page 415
- “\_debug\_memset - Set Bytes to Value” on page 416
- “\_debug\_realloc - Reallocate Memory Block” on page 417
- “\_debug\_strcat - Concatenate Strings” on page 419
- “\_debug\_strcpy - Copy Strings” on page 421
- “\_debug\_strncat - Concatenate Strings” on page 422
- “\_debug\_strncpy - Copy Strings” on page 423
- “\_debug\_strnset - Set Characters in String” on page 425
- “\_debug\_strset - Set Characters in String” on page 426
- “\_debug\_ucalloc - Reserve and Initialize Memory from User Heap” on page 428
- “\_debug\_uheapmin - Free Unused Memory in User Heap” on page 430
- “\_debug\_umalloc - Reserve Memory Blocks from User Heap” on page 431
- “heapdebug” on page 270

---

## **`_debug_memcpy` - Copy Bytes**

### Format

```

#include <string.h>
void *_debug_memcpy(void *dest, const void *src, size_t count,
                   const char *file, size_t line);

```

**Language Level:** Extension

`_debug_memcpy` is the debug version of `memcpy`. Like `memcpy`, it copies `count` bytes of `src` to `dest`, where the behavior is undefined if copying takes place between objects that overlap.

`_debug_memcpy` validates the heap after copying the bytes to the target location, and performs this check only when the target is within a heap. `_debug_memcpy` makes an implicit call to `_heap_check`. If `_debug_memcpy` detects a corrupted heap when it makes a call to `_heap_check`, `_debug_memcpy` will report the file name `file` and line number `line` in a message.

**Note:** `_debug_memcpy` checks only the current default heap. Therefore, this debug support will not check all heaps within applications that have multiple user heaps.

To use `_debug_memcpy`, you must compile with the debug memory `-qheapdebug` compiler option. This option maps all `memcpy` calls to `_debug_memcpy`. You do not have to change your source code, in order for `_debug_memcpy` to verify the heap.

**Note:** The **-qheapdebug** option maps all calls to other string functions and all calls to memory management functions (including a heap-specific version), to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

### Return Value

**\_debug\_memcpy** returns a pointer to *dest*.

### Example

This example contains a programming error. On the **memcpy** used to initialize the target location, the count is more than the size of the target object, and the **memcpy** operation copies bytes past the end of the allocated object.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#define MAX_LEN      10
int main(void)
{
    char *source, *target;
    target = (char*)malloc(MAX_LEN);
    memcpy(target, "This is the target string", 11);
    printf("Target is \"%s\"\n", target);
    return 0;

    /*****
    The output should be similar to:
        End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
        The first eight bytes of the memory block (in hex) are: 5468697320697320.
        This memory block was (re)allocated at line number 11 in memcpy.c.
        Heap state was valid at line 11 of memcpy.c.
        Memory error detected at line 12 of memcpy.c.
    *****/
}
```

### Related Reading

- “Debugging Memory Heaps” on page 183
- “Memory Management Functions” on page 179
- “Managing Memory with Multiple Heaps” on page 182
- “Debugging Programs with Heap Memory” on page 195
- “heapdebug” on page 270
- “\_debug\_calloc - Allocate and Initialize Memory” on page 407
- “\_debug\_free - Free Allocated Memory” on page 408
- “\_debug\_heapmin - Free Unused Memory in the Default Heap” on page 410
- “\_debug\_malloc - Allocate Memory” on page 412
- “\_debug\_memmove - Copy Bytes” on page 415
- “\_debug\_memset - Set Bytes to Value” on page 416
- “\_debug\_realloc - Reallocate Memory Block” on page 417
- “\_debug\_strcat - Concatenate Strings” on page 419
- “\_debug\_strcpy - Copy Strings” on page 421
- “\_debug\_strncat - Concatenate Strings” on page 422
- “\_debug\_strncpy - Copy Strings” on page 423
- “\_debug\_strnset - Set Characters in String” on page 425
- “\_debug\_strset - Set Characters in String” on page 426
- “\_debug\_ucalloc - Reserve and Initialize Memory from User Heap” on page 428
- “\_debug\_uheapmin - Free Unused Memory in User Heap” on page 430
- “\_debug\_umalloc - Reserve Memory Blocks from User Heap” on page 431
- “heapdebug” on page 270

---

## **`_debug_memmove` - Copy Bytes**

### **Format**

```
#include <string.h>
void *_debug_memmove(void *dest, const void *src, size_t count,
                    const char *file, size_t line);
```

### **Language Level:** Extension

`_debug_memmove` is the debug version of `memmove`. Like `memmove`, it copies *count* bytes of *src* to *dest*, and allows for copying between objects that may overlap.

`_debug_memmove` validates the heap after copying the bytes to the target location, and performs this check only when the target is within a heap. `_debug_memmove` makes an implicit call to `_heap_check`. If `_debug_memmove` detects a corrupted heap when it makes a call to `_heap_check`, `_debug_memmove` will report the file name *file* and line number *line* in a message.

**Note:** `_debug_memmove` checks only the current default heap. Therefore, this debug support will not check all heaps within applications that have multiple user heaps.

To use `_debug_memmove`, you must compile with the debug memory `-qheapdebug` compiler option. This option maps all `memcpy` calls to `_debug_memmove`. You do not have to change your source code, in order for `_debug_memmove` to verify the heap.

**Note:** The `-qheapdebug` option maps all calls to other string functions and all calls to memory management functions (including a heap-specific version), to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

### **Return Value**

`_debug_memmove` returns a pointer to *dest*.

### **Example**

This example contains a programming error. The count specified on `memmove` is 15 instead of 5, and the `memmove` operation copies bytes past the end of the allocated object.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define SIZE          21

int main(void)
{
    char *target, *p, *source;
    target = (char*)malloc(SIZE);
    strcpy(target, "a shiny white sphere");
    p = target+8;
    source = target+2;
    printf("Before memmove, target is \"%s\"\n", target);
    memmove(p, source, 15);
    printf("After memmove, target becomes \"%s\"\n", target);
    return 0;
}

/*****
The output should be similar to:
Before memmove, target is "a shiny white sphere"
End of allocated object 0x00073c80 was overwritten at 0x00073c95.
The first eight bytes of the memory block (in hex) are: 61207368696E7920.
This memory block was (re)allocated at line number 11 in memmove.c.
*****/
```

```

    Heap state was valid at line 12 of memmove.c.
    Memory error detected at line 18 of memcpy.c.
    *****/
}

```

## Related Reading

- “Debugging Memory Heaps” on page 183
- “Memory Management Functions” on page 179
- “Managing Memory with Multiple Heaps” on page 182
- “Debugging Programs with Heap Memory” on page 195
- “heapdebug” on page 270
- “\_debug\_calloc - Allocate and Initialize Memory” on page 407
- “\_debug\_free - Free Allocated Memory” on page 408
- “\_debug\_heapmin - Free Unused Memory in the Default Heap” on page 410
- “\_debug\_malloc - Allocate Memory” on page 412
- “\_debug\_memcpy - Copy Bytes” on page 413
- “\_debug\_memset - Set Bytes to Value”
- “\_debug\_realloc - Reallocate Memory Block” on page 417
- “\_debug\_strcat - Concatenate Strings” on page 419
- “\_debug\_strcpy - Copy Strings” on page 421
- “\_debug\_strncat - Concatenate Strings” on page 422
- “\_debug\_strncpy - Copy Strings” on page 423
- “\_debug\_strnset - Set Characters in String” on page 425
- “\_debug\_strset - Set Characters in String” on page 426
- “\_debug\_ucalloc - Reserve and Initialize Memory from User Heap” on page 428
- “\_debug\_uheapmin - Free Unused Memory in User Heap” on page 430
- “\_debug\_umalloc - Reserve Memory Blocks from User Heap” on page 431
- “heapdebug” on page 270

---

## **\_debug\_memset - Set Bytes to Value**

### Format

```

#include <string.h>
void *_debug_memset(void *dest, int c, size_t count,
                   const char *file, size_t line);

```

**Language Level:** Extension

**\_debug\_memset** is the debug version of **memset**. Like **memset**, it sets the first *count* bytes of *dest* to the value *c*. The value of *c* is converted to an unsigned character.

**\_debug\_memset** validates the heap after setting the bytes, and performs this check only when the target is within a heap. **\_debug\_memset** makes an implicit call to **\_heap\_check**. If **\_debug\_memset** detects a corrupted heap when it makes a call to **\_heap\_check**, **\_debug\_memset** will report the file name *file* and line number *line* in a message.

**Note:** **\_debug\_memset** checks only the current default heap. Therefore, this debug support will not check all heaps within applications that have multiple user heaps.

To use **\_debug\_memset**, you must compile with the debug memory **-qheapdebug** compiler option. This option maps all **memset** calls to **\_debug\_memset**. You do not have to change your source code, in order for **\_debug\_memset** to verify the heap.

**Note:** The **-qheapdebug** option maps all calls to other string functions and all calls to memory management functions (including a heap-specific version), to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

## Return Value

`_debug_memset` returns a pointer to *dest*.

## Example

This example contains a programming error. The invocation of `memset` that puts 'B' in the buffer specifies the wrong count, and stores bytes past the end of the buffer.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#define BUF_SIZE 20
int main(void)
{
    char *buffer, *buffer2;
    char *string;
    buffer = (char*)calloc(1, BUF_SIZE+1); /* +1 for null-terminator */
    string = (char*)memset(buffer, 'A', 10);
    printf("\nBuffer contents: %s\n", string);
    memset(buffer+10, 'B', 20);
    return 0;
    /*****
    The output should be:
    Buffer contents: AAAAAAAAAA
    End of allocated object 0x00073c80 was overwritten at 0x00073c95.
    The first eight bytes of the memory block (in hex) are: 4141414141414141.
    This memory block was (re)allocated at line number 12 in memset.c.
    Heap state was valid at line 14 of memset.c.
    Memory error detected at line 16 of memset.c.
    *****/
}
```

## Related Reading

- “Debugging Memory Heaps” on page 183
- “Memory Management Functions” on page 179
- “Managing Memory with Multiple Heaps” on page 182
- “Debugging Programs with Heap Memory” on page 195
- “heapdebug” on page 270
- “`_debug_calloc` - Allocate and Initialize Memory” on page 407
- “`_debug_free` - Free Allocated Memory” on page 408
- “`_debug_heapmin` - Free Unused Memory in the Default Heap” on page 410
- “`_debug_malloc` - Allocate Memory” on page 412
- “`_debug_memcpy` - Copy Bytes” on page 413
- “`_debug_memmove` - Copy Bytes” on page 415
- “`_debug_realloc` - Reallocate Memory Block”
- “`_debug_strcat` - Concatenate Strings” on page 419
- “`_debug_strcpy` - Copy Strings” on page 421
- “`_debug_strncat` - Concatenate Strings” on page 422
- “`_debug_strncpy` - Copy Strings” on page 423
- “`_debug_strnset` - Set Characters in String” on page 425
- “`_debug_strset` - Set Characters in String” on page 426
- “`_debug_ucalloc` - Reserve and Initialize Memory from User Heap” on page 428
- “`_debug_uheapmin` - Free Unused Memory in User Heap” on page 430
- “`_debug_umalloc` - Reserve Memory Blocks from User Heap” on page 431

---

## `_debug_realloc` - Reallocate Memory Block

### Format

```
#include <stdlib.h> /* also in <malloc.h> */
void *_debug_realloc(void *ptr, size_t size,
                    const char *file, size_t line);
```

## Language Level: Extension

**\_debug\_realloc** is the debug version of **realloc**. Like **realloc**, it reallocates the block of memory pointed to by *ptr* to a new *size*, specified in bytes. It also sets any new memory it allocates to 0xAA, so you can easily locate instances where your program tries to use the data in that memory without initializing it first.

In addition, **\_debug\_realloc** makes an implicit call to **\_heap\_check**, and stores the file name *file* and the line number *line* where the storage is reallocated. This information can be used later by the **\_heap\_check**, **\_dump\_allocated**, or **\_dump\_allocated\_delta** functions.

If *ptr* is NULL, **\_debug\_realloc** behaves like **\_debug\_malloc** (or **malloc**) and allocates the block of memory.

To use **\_debug\_realloc**, you must compile with the debug memory **-qheapdebug** compiler option. This option maps all **realloc** calls to **\_debug\_realloc**.

**Note:** The **-qheapdebug** option maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Because **\_debug\_realloc** always checks what heap the memory was allocated from, you can use **\_debug\_realloc** to reallocate memory blocks allocated by the regular or debug versions of the memory management functions. However, if the memory was not allocated by the memory management functions, or was previously freed, **\_debug\_realloc** generates an error message and the program ends.

## Return Value

**\_debug\_realloc** returns a pointer to the reallocated memory block. The *ptr* argument to **\_debug\_realloc** is not the same as the return value; **\_debug\_realloc** always changes the memory location to help you locate references to the memory that were not freed before the memory was reallocated.

If *size* is 0, **\_debug\_realloc** returns NULL. If not enough memory is available to expand the block to the given size, the original block is unchanged and NULL is returned.

## Example

This example uses **\_debug\_realloc** to allocate 100 bytes of storage. It then attempts to write to storage that was not allocated. When **\_debug\_realloc** is called again, **\_heap\_check** detects the error, generates several messages, and stops the program.

**Note:** You must compile this example with the **-qheapdebug** option to map the **realloc** calls to **\_debug\_realloc**.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *ptr;
    if (NULL == (ptr = (char*)realloc(NULL, 100))) {
        puts("Could not allocate memory block.");
        exit(EXIT_FAILURE);
    }
    memset(ptr, 'a', 105); /* overwrites storage that was not allocated */
    ptr = (char*)realloc(ptr, 200); /* realloc invokes _heap_check */
    puts("_debug_realloc did not detect that a memory block was overwritten.");
    return 0;
}
```

```

/*****
The output should be similar to:
End of allocated object 0x00073890 was overwritten at 0x000738f4.
The first eight bytes of the memory block (in hex) are: 6161616161616161.
This memory block was (re)allocated at line number 8 in _debug_reall.c.
Heap state was valid at line 8 of _debug_reall.c.
Memory error detected at line 13 of _debug_reall.c.
*****/
}

```

### Related Reading

- “Debugging Memory Heaps” on page 183
- “Memory Management Functions” on page 179
- “Managing Memory with Multiple Heaps” on page 182
- “Debugging Programs with Heap Memory” on page 195
- “heapdebug” on page 270
- “\_debug\_calloc - Allocate and Initialize Memory” on page 407
- “\_debug\_free - Free Allocated Memory” on page 408
- “\_debug\_heapmin - Free Unused Memory in the Default Heap” on page 410
- “\_debug\_malloc - Allocate Memory” on page 412
- “\_debug\_memcpy - Copy Bytes” on page 413
- “\_debug\_memmove - Copy Bytes” on page 415
- “\_debug\_memset - Set Bytes to Value” on page 416
- “\_debug\_strcat - Concatenate Strings”
- “\_debug\_strcpy - Copy Strings” on page 421
- “\_debug\_strncat - Concatenate Strings” on page 422
- “\_debug\_strncpy - Copy Strings” on page 423
- “\_debug\_strnset - Set Characters in String” on page 425
- “\_debug\_strset - Set Characters in String” on page 426
- “\_debug\_ucalloc - Reserve and Initialize Memory from User Heap” on page 428
- “\_debug\_uheapmin - Free Unused Memory in User Heap” on page 430
- “\_debug\_umalloc - Reserve Memory Blocks from User Heap” on page 431
- “heapdebug” on page 270

---

## **\_\_debug\_strcat - Concatenate Strings**

### Format

```

#include <string.h>
char *__debug_strcat(char *string1, const char *string2,
                    const char *file, size_t file);

```

### Language Level:Extension

**\_\_debug\_strcat** is the debug version of **strcat**. Like **strcat**, it concatenates *string2* to *string1* and ends the resulting string with the null character.

**\_\_debug\_strcat** validates the heap after concatenating the strings, and performs this check only when the target is within a heap. **\_\_debug\_strcat** makes an implicit call to **\_\_heap\_check**. If **\_\_debug\_strcat** detects a corrupted heap when it makes a call to **\_\_heap\_check**, **\_\_debug\_strcat** will report the file name *file* and line number *file* in a message.

**Note:** **\_\_debug\_strcat** checks only the current default heap. Therefore, this debug support will not check all heaps within applications that have multiple user heaps.

To use **\_\_debug\_strcat**, you must compile with the debug memory **-qheapdebug** compiler option. This option maps all **strcat** calls to **\_\_debug\_strcat**. You do not have to change your source code, in order for **\_\_debug\_strcat** to verify the heap.



**Note:** The **-qheapdebug** option maps all calls to other string functions and all calls to memory management functions (including a heap-specific version), to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

## Return Value

**\_debug\_strcat** returns a pointer to the concatenated string *string1*.

## Example

This example contains a programming error. The `buffer1` object is not large enough to store the result after the string " program" is concatenated.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define SIZE 10
int main(void)
{
    char *buffer1;
    char *ptr;

    buffer1 = (char*)malloc(SIZE);
    strcpy(buffer1, "computer");

    ptr = strcat(buffer1, " program");
    printf("buffer1 = %s\n", buffer1);
    return 0;

    /*****
    The output should be similar to:

    End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
    The first eight bytes of the memory block (in hex) are: 636F6D7075746572.
    This memory block was (re)allocated at line number 12 in strcat.c.
    Heap state was valid at line 13 of strcat.c.
    Memory error detected at line 15 of strcat.c.
    *****/
}
```

## Related Reading

- “Debugging Memory Heaps” on page 183
- “Memory Management Functions” on page 179
- “Managing Memory with Multiple Heaps” on page 182
- “Debugging Programs with Heap Memory” on page 195
- “heapdebug” on page 270
- “\_debug\_calloc - Allocate and Initialize Memory” on page 407
- “\_debug\_free - Free Allocated Memory” on page 408
- “\_debug\_heapmin - Free Unused Memory in the Default Heap” on page 410
- “\_debug\_malloc - Allocate Memory” on page 412
- “\_debug\_memcpy - Copy Bytes” on page 413
- “\_debug\_memmove - Copy Bytes” on page 415
- “\_debug\_memset - Set Bytes to Value” on page 416
- “\_debug\_realloc - Reallocate Memory Block” on page 417
- “\_debug\_strcpy - Copy Strings” on page 421
- “\_debug\_strncat - Concatenate Strings” on page 422
- “\_debug\_strncpy - Copy Strings” on page 423
- “\_debug\_strnset - Set Characters in String” on page 425
- “\_debug\_strset - Set Characters in String” on page 426
- “\_debug\_ucalloc - Reserve and Initialize Memory from User Heap” on page 428
- “\_debug\_uheapmin - Free Unused Memory in User Heap” on page 430
- “\_debug\_umalloc - Reserve Memory Blocks from User Heap” on page 431
- “heapdebug” on page 270



---

## **\_\_debug\_strcpy - Copy Strings**

### **Format**

```
#include <string.h>
char *__debug_strcpy(char *string1, const char *string2,
                    const char *file, size_t line);
```

### **Language Level:** Extension

**\_\_debug\_strcpy** is the debug version of **strcpy**. Like **strcpy**, it copies *string2*, including the ending null character, to the location specified by *string1*.

**\_\_debug\_strcpy** validates the heap after copying the string to the target location, and performs this check only when the target is within a heap. **\_\_debug\_strcpy** makes an implicit call to **\_\_heap\_check**. If **\_\_debug\_strcpy** detects a corrupted heap when it makes a call to **\_\_heap\_check**, **\_\_debug\_strcpy** will report the file name *file* and line number *line* in a message.

**Note:** **\_\_debug\_strcpy** checks only the current default heap. Therefore, this debug support will not check all heaps within applications that have multiple user heaps.

To use **\_\_debug\_strcpy**, you must compile with the debug memory **-qheapdebug** compiler option. This option maps all **strcpy** calls to **\_\_debug\_strcpy**. You do not have to change your source code, in order for **\_\_debug\_strcpy** to verify the heap.

**Note:** The **-qheapdebug** option maps all calls to other string functions and all calls to memory management functions (including a heap-specific version), to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

### **Return Value**

**\_\_debug\_strcpy** returns a pointer to the copied string *string1*.

### **Example**

This example contains a programming error. The source string is too long for the destination buffer, and the **strcpy** operation damages the heap.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define SIZE          10

int main(void)
{
    char *source = "1234567890123456789";
    char *destination;
    char *return_string;

    destination = (char*)malloc(SIZE);
    strcpy(destination, "abcdefg"),
    printf("destination is originally = '%s'\n", destination);
    return_string = strcpy(destination, source);
    printf("After strcpy, destination becomes '%s'\n\n", destination);
    return 0;
}

/*****
The output should be similar to:
destination is originally = 'abcdefg'
End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
The first eight bytes of the memory block (in hex) are: 3132333435363738.
This memory block was (re)allocated at line number 13 in strcpy.c.
*****/
```

```

    Heap state was valid at line 14 of strcpy.c.
    Memory error detected at line 17 of strcpy.c.
    *****/
}

```

## Related Reading

- “Debugging Memory Heaps” on page 183
- “Memory Management Functions” on page 179
- “Managing Memory with Multiple Heaps” on page 182
- “Debugging Programs with Heap Memory” on page 195
- “heapdebug” on page 270
- “\_debug\_calloc - Allocate and Initialize Memory” on page 407
- “\_debug\_free - Free Allocated Memory” on page 408
- “\_debug\_heapmin - Free Unused Memory in the Default Heap” on page 410
- “\_debug\_malloc - Allocate Memory” on page 412
- “\_debug\_memcpy - Copy Bytes” on page 413
- “\_debug\_memmove - Copy Bytes” on page 415
- “\_debug\_memset - Set Bytes to Value” on page 416
- “\_debug\_realloc - Reallocate Memory Block” on page 417
- “\_debug\_strcat - Concatenate Strings” on page 419
- “\_debug\_strncat - Concatenate Strings”
- “\_debug\_strncpy - Copy Strings” on page 423
- “\_debug\_strnset - Set Characters in String” on page 425
- “\_debug\_strset - Set Characters in String” on page 426
- “\_debug\_ucalloc - Reserve and Initialize Memory from User Heap” on page 428
- “\_debug\_uheapmin - Free Unused Memory in User Heap” on page 430
- “\_debug\_umalloc - Reserve Memory Blocks from User Heap” on page 431
- “heapdebug” on page 270

---

## **\_debug\_strncat - Concatenate Strings**

### Format

```

#include <string.h>
char *_debug_strncat(char *string1, const char *string2, size_t count,
                    const char *file, size_t line);

```

**Language Level:** Extension

**\_debug\_strncat** is the debug version of **strncat**. Like **strncat**, it appends the first *count* characters of *string2* to *string1* and ends the resulting string with a null character (`\0`). If *count* is greater than the length of *string2*, the length of *string2* is used in place of *count*.

**\_debug\_strncat** validates the heap after appending the characters, and performs this check only when the target is within a heap. **\_debug\_strncat** makes an implicit call to **\_heap\_check**. If **\_debug\_strncat** detects a corrupted heap when it makes a call to **\_heap\_check**, **\_debug\_strncat** will report the file name *file* and line number *line* in a message.

**Note:** **\_debug\_strncat** checks only the current default heap. Therefore, this debug support will not check all heaps within applications that have multiple user heaps.

To use **\_debug\_strncat**, you must compile with the debug memory **-qheapdebug** compiler option. This option maps all **strncat** calls to **\_debug\_strncat**. You do not have to change your source code, in order for **\_debug\_strncat** to verify the heap.

**Note:** The **-qheapdebug** option maps all calls to other string functions and all calls to memory management functions (including a heap-specific version), to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

## Return Value

`_debug_strncat` returns a pointer to the joined string *string1*.

## Example

This example contains a programming error. The `buffer1` object is not large enough to store the result after eight characters from the string " programming" are concatenated.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define SIZE          10

int main(void)
{
    char *buffer1;
    char *ptr;

    buffer1 = (char*)malloc(SIZE);
    strcpy(buffer1, "computer");

    /* Call strncat with buffer1 and " programming" */
    ptr = strncat(buffer1, " programming", 8);
    printf("strncat: buffer1 = \"%s\\n\"", buffer1);
    return 0;

    /*****
    The output should be similar to:

    End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
    The first eight bytes of the memory block (in hex) are: 636F6D7075746572.
    This memory block was (re)allocated at line number 12 in strncat.c.
    Heap state was valid at line 13 of strncat.c.
    Memory error detected at line 17 of strncat.c.
    *****/
}
```

## Related Reading

- “Debugging Memory Heaps” on page 183
- “Memory Management Functions” on page 179
- “Managing Memory with Multiple Heaps” on page 182
- “Debugging Programs with Heap Memory” on page 195
- “heapdebug” on page 270
- “`_debug_calloc` - Allocate and Initialize Memory” on page 407
- “`_debug_free` - Free Allocated Memory” on page 408
- “`_debug_heapmin` - Free Unused Memory in the Default Heap” on page 410
- “`_debug_malloc` - Allocate Memory” on page 412
- “`_debug_memcpy` - Copy Bytes” on page 413
- “`_debug_memmove` - Copy Bytes” on page 415
- “`_debug_memset` - Set Bytes to Value” on page 416
- “`_debug_realloc` - Reallocate Memory Block” on page 417
- “`_debug_strcat` - Concatenate Strings” on page 419
- “`_debug_strcpy` - Copy Strings” on page 421
- “`_debug_strncpy` - Copy Strings”
- “`_debug_strnset` - Set Characters in String” on page 425
- “`_debug_strset` - Set Characters in String” on page 426
- “`_debug_ucalloc` - Reserve and Initialize Memory from User Heap” on page 428
- “`_debug_uheapmin` - Free Unused Memory in User Heap” on page 430
- “`_debug_umalloc` - Reserve Memory Blocks from User Heap” on page 431
- “heapdebug” on page 270

---

## `_debug_strncpy` - Copy Strings

### Format

```
#include <string.h>
char *_debug_strncpy(char *string1, const char *string2, size_t count,
                    const char *file, size_t line);
```

## Language Level:Extension

**\_debug\_strncpy** is the debug version of **strncpy**. Like **strncpy**, it copies *count* characters of *string2* to *string1*. If *count* is less than or equal to the length of *string2*, a null character (\0) is not appended to the copied string. If *count* is greater than the length of *string2*, the *string1* result is padded with null characters (\0) up to length *count*.

**\_debug\_strncpy** validates the heap after copying the strings to the target location, and performs this check only when the target is within a heap. **\_debug\_strncpy** makes an implicit call to **\_heap\_check**. If **\_debug\_strncpy** detects a corrupted heap when it makes a call to **\_heap\_check**, **\_debug\_strncpy** will report the file name *file* and line number *line* in a message.

**Note:** **\_debug\_strncpy** checks only the current default heap. Therefore, this debug support will not check all heaps within applications that have multiple user heaps.

To use **\_debug\_strncpy**, you must compile with the debug memory **-qheapdebug** compiler option. This option maps all **strncpy** calls to **\_debug\_strncpy**. You do not have to change your source code, in order for **\_debug\_strncpy** to verify the heap.

**Note:** The **-qheapdebug** option maps all calls to other string functions and all calls to memory management functions (including a heap-specific version), to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

## Return Value

**\_debug\_strncpy** returns a pointer to *string1*.

## Example

This example contains a programming error. The source string is too long for the destination buffer, and the **strncpy** operation damages the heap.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define SIZE          10
int main(void)
{
    char *source = "1234567890123456789";
    char *destination;
    char *return_string;
    int index = 15;
    destination = (char*)malloc(SIZE);
    strncpy(destination, "abcdefg"),
    printf("destination is originally = '%s'\n", destination);
    return_string = strncpy(destination, source, index);
    printf("After strncpy, destination becomes '%s'\n\n", destination);
    return 0;
}
/*****
The output should be similar to:
destination is originally = 'abcdefg'
End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
The first eight bytes of the memory block (in hex) are: 3132333435363738.
This memory block was (re)allocated at line number 14 in strncpy.c.
*****/
```

```

    Heap state was valid at line 15 of strncpy.c.
    Memory error detected at line 18 of strncpy.c.
    *****/
}

```

### Related Reading

- “Debugging Memory Heaps” on page 183
- “Memory Management Functions” on page 179
- “Managing Memory with Multiple Heaps” on page 182
- “Debugging Programs with Heap Memory” on page 195
- “heapdebug” on page 270
- “\_debug\_calloc - Allocate and Initialize Memory” on page 407
- “\_debug\_free - Free Allocated Memory” on page 408
- “\_debug\_heapmin - Free Unused Memory in the Default Heap” on page 410
- “\_debug\_malloc - Allocate Memory” on page 412
- “\_debug\_memcpy - Copy Bytes” on page 413
- “\_debug\_memmove - Copy Bytes” on page 415
- “\_debug\_memset - Set Bytes to Value” on page 416
- “\_debug\_realloc - Reallocate Memory Block” on page 417
- “\_debug\_strcat - Concatenate Strings” on page 419
- “\_debug\_strcpy - Copy Strings” on page 421
- “\_debug\_strncat - Concatenate Strings” on page 422
- “\_debug\_strnset - Set Characters in String”
- “\_debug\_strset - Set Characters in String” on page 426
- “\_debug\_ucalloc - Reserve and Initialize Memory from User Heap” on page 428
- “\_debug\_uheapmin - Free Unused Memory in User Heap” on page 430
- “\_debug\_umalloc - Reserve Memory Blocks from User Heap” on page 431
- “heapdebug” on page 270

---

## **\_debug\_strnset - Set Characters in String**

### Format

```

#include <string.h>
char *_debug_strnset(char *string, int c, size_t n,
                    const char *file, size_t line);

```

### Language Level:Extension

**\_debug\_strnset** is the debug version of **strnset**. Like **strnset**, it sets, at most, the first *n* characters of *string* to *c* (converted to a char), where if *n* is greater than the length of *string*, the length of *string* is used in place of *n*.

**\_debug\_strnset** validates the heap after setting the bytes, and performs this check only when the target is within a heap. **\_debug\_strnset** makes an implicit call to **\_heap\_check**. If **\_debug\_strnset** detects a corrupted heap when it makes a call to **\_heap\_check**, **\_debug\_strnset** will report the file name *file* and line number *line* in a message.

**Note:** **\_debug\_strnset** checks only the current default heap. Therefore, this debug support will not check all heaps within applications that have multiple user heaps.

To use **\_debug\_strnset**, you must compile with the debug memory **-qheapdebug** compiler option. This option maps all **strnset** calls to **\_debug\_strnset**. You do not have to change your source code, in order for **\_debug\_memset** to verify the heap.

**Note:** The **-qheapdebug** option maps all calls to memory management functions (including a heap-specific version) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

## Return Value

`_debug_strnset` returns a pointer to the altered *string*. There is no error return value.

## Example

This example contains two programming errors. The string, *str*, was created without a null-terminator to mark the end of the string, and without the terminator `strnset` with a count of 10 stores bytes past the end of the allocated object.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *str;
    str = (char*)malloc(10);
    printf("This is the string after strnset: %s\n", str);
    return 0;

    /*****
    The output should be:
    End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
    The first eight bytes of the memory block (in hex) are: 7878787878797979.
    This memory block was (re)allocated at line number 9 in strnset.c.
    Heap state was valid at line 11 of strnset.c.
    *****/
}
```

## Related Reading

- “Debugging Memory Heaps” on page 183
- “Memory Management Functions” on page 179
- “Managing Memory with Multiple Heaps” on page 182
- “Debugging Programs with Heap Memory” on page 195
- “heapdebug” on page 270
- “\_debug\_calloc - Allocate and Initialize Memory” on page 407
- “\_debug\_free - Free Allocated Memory” on page 408
- “\_debug\_heapmin - Free Unused Memory in the Default Heap” on page 410
- “\_debug\_malloc - Allocate Memory” on page 412
- “\_debug\_memcpy - Copy Bytes” on page 413
- “\_debug\_memmove - Copy Bytes” on page 415
- “\_debug\_memset - Set Bytes to Value” on page 416
- “\_debug\_realloc - Reallocate Memory Block” on page 417
- “\_debug\_strcat - Concatenate Strings” on page 419
- “\_debug\_strcpy - Copy Strings” on page 421
- “\_debug\_strncat - Concatenate Strings” on page 422
- “\_debug\_strncpy - Copy Strings” on page 423
- “\_debug\_strset - Set Characters in String”
- “\_debug\_ucalloc - Reserve and Initialize Memory from User Heap” on page 428
- “\_debug\_uheapmin - Free Unused Memory in User Heap” on page 430
- “\_debug\_umalloc - Reserve Memory Blocks from User Heap” on page 431
- “heapdebug” on page 270

---

## `_debug_strset` - Set Characters in String

### Format

```
#include <string.h>
char *_debug_strset(char *string, size_t c,
                   const char *file, size_t line);
```

**Language Level:**Extension

`_debug_strset` is the debug version of `strset`. Like `strset`, it sets all characters of *string*, except the ending null character (`\0`), to *c* (converted to a `char`).

`_debug_strset` validates the heap after setting all characters of *string*, and performs this check only when the target is within a heap. `_debug_strset` makes an implicit call to `_heap_check`. If `_debug_strset` detects a corrupted heap when it makes a call to `_heap_check`, `_debug_strset` will report the file name *file* and line number *line* in a message.

**Note:** `_debug_strset` checks only the current default heap. Therefore, this debug support will not check all heaps within applications that have multiple user heaps.

To use `_debug_strset`, you must compile with the debug memory `-qheapdebug` compiler option. This option maps all `strset` calls to `_debug_strset`. You do not have to change your source code, in order for `_debug_strset` to verify the heap.

**Note:** The `-qheapdebug` option maps all calls to other string functions and all calls to memory management functions (including a heap-specific version), to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

### Return Value

`_debug_strset` returns a pointer to the altered *string*. There is no error return value.

### Example

This example contains a programming error. The string, *str*, was created without a null-terminator, and `strset` propagates the letter 'k' until it finds what it thinks is the null-terminator.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *str;
    str = (char*)malloc(10);
    strnset(str, 'x', 5);
    strset(str+5, 'k');
    printf("This is the string after strset: %s\n", str);
    return 0;
}

/*****
    The output should be:
    End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
    The first eight bytes of the memory block (in hex) are: 787878786B6B6B.
    This memory block was (re)allocated at line number 9 in strset.c.
    Heap state was valid at line 11 of strset.c.
    Memory error detected at line 12 of strset.c.
*****/
```



## Related Reading

“Debugging Memory Heaps” on page 183  
“Memory Management Functions” on page 179  
“Managing Memory with Multiple Heaps” on page 182  
“Debugging Programs with Heap Memory” on page 195  
“heapdebug” on page 270  
“\_debug\_calloc - Allocate and Initialize Memory” on page 407  
“\_debug\_free - Free Allocated Memory” on page 408  
“\_debug\_heapmin - Free Unused Memory in the Default Heap” on page 410  
“\_debug\_malloc - Allocate Memory” on page 412  
“\_debug\_memcpy - Copy Bytes” on page 413  
“\_debug\_memmove - Copy Bytes” on page 415  
“\_debug\_memset - Set Bytes to Value” on page 416  
“\_debug\_realloc - Reallocate Memory Block” on page 417  
“\_debug\_strcat - Concatenate Strings” on page 419  
“\_debug\_strcpy - Copy Strings” on page 421  
“\_debug\_strncat - Concatenate Strings” on page 422  
“\_debug\_strncpy - Copy Strings” on page 423  
“\_debug\_strnset - Set Characters in String” on page 425  
“\_debug\_ucalloc - Reserve and Initialize Memory from User Heap”  
“\_debug\_uheapmin - Free Unused Memory in User Heap” on page 430  
“\_debug\_umalloc - Reserve Memory Blocks from User Heap” on page 431  
“heapdebug” on page 270

---

## **\_debug\_ucalloc - Reserve and Initialize Memory from User Heap**

### Format

```
#include <umalloc.h>
void *_debug_ucalloc(Heap_t heap, size_t num, size_t size,
                    const char *file, size_t line);
```

**Language Level:** Extension

**\_debug\_ucalloc** is the debug version of **\_ucalloc**. Like **\_ucalloc**, it allocates memory from the *heap* you specify for an array of *num* elements, each of length *size* bytes. It then initializes all bits of each element to 0.

In addition, **\_debug\_ucalloc** makes an implicit call to **\_uheap\_check**, and stores the name of the file *file* and the line number *line* where the storage is allocated. This information can be used later by the **\_uheap\_check**, **\_uheap\_allocated**, or **\_udump\_allocated\_delta** functions.

To use **\_debug\_ucalloc**, you must compile with the debug memory **-qheapdebug** compiler option. This option maps all **\_ucalloc** calls to **\_debug\_ucalloc**.

**Note:** The **-qheapdebug** option maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

**\_debug\_ucalloc** works just like **\_debug\_calloc** except that you specify the heap to use; **\_debug\_calloc** always allocates from the default heap.

If the *heap* does not have enough memory for the request, **\_debug\_ucalloc** calls the *getmore\_fn* that you specified when you created the heap with **\_ucreate**.

To reallocate or free memory allocated with **\_debug\_ucalloc**, use the non-heap-specific **\_debug\_realloc** and **\_debug\_free**. These functions always check what heap the memory was allocated from.



## Return Value

`_debug_ucalloc` returns a pointer to the reserved space. If *size* or *num* was specified as zero, or if your *getmore\_fn* cannot provide enough memory, `_debug_ucalloc` returns NULL. Passing `_debug_ucalloc` a heap that is not valid results in undefined behavior.

## Example

This example creates a user heap and allocates memory from it with `_debug_ucalloc`. It then attempts to write to memory that was not allocated. When `_debug_free` is called, `_uheap_check` detects the error, generates several messages, and stops the program.

**Note:** You must compile this example with the `-qheapdebug` option to map the `_ucalloc` calls to `_debug_ucalloc` and free to `_debug_free`.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>
#include <string.h>
int main(void)
{
    Heap_t myheap;
    char *ptr;

    /* Use default heap as user heap */
    myheap = _udefault(NULL);
    if (NULL == (ptr = (char*)_ucalloc(myheap, 100, 1))) {
        puts("Cannot allocate memory from user heap.");
        exit(EXIT_FAILURE);
    }
    memset(ptr, 'x', 105); /* Overwrites storage that was not allocated */
    free(ptr);
    return 0;

    /*****
    The output should be similar to :

    End of allocated object 0x00073890 was overwritten at 0x000738f4.
    The first eight bytes of the memory block (in hex) are: 7878787878787878.
    This memory block was (re)allocated at line number 14 in _debug_ucallo.c.
    Heap state was valid at line 14 of _debug_ucallo.c.
    Memory error detected at line 19 of _debug_ucallo.c.
    *****/
}
```

## Related Reading

“Debugging Memory Heaps” on page 183  
“Memory Management Functions” on page 179  
“Managing Memory with Multiple Heaps” on page 182  
“Debugging Programs with Heap Memory” on page 195  
“heapdebug” on page 270  
“\_debug\_calloc - Allocate and Initialize Memory” on page 407  
“\_debug\_free - Free Allocated Memory” on page 408  
“\_debug\_heapmin - Free Unused Memory in the Default Heap” on page 410  
“\_debug\_malloc - Allocate Memory” on page 412  
“\_debug\_memcpy - Copy Bytes” on page 413  
“\_debug\_memmove - Copy Bytes” on page 415  
“\_debug\_memset - Set Bytes to Value” on page 416  
“\_debug\_realloc - Reallocate Memory Block” on page 417  
“\_debug\_strcat - Concatenate Strings” on page 419  
“\_debug\_strcpy - Copy Strings” on page 421  
“\_debug\_strncat - Concatenate Strings” on page 422  
“\_debug\_strncpy - Copy Strings” on page 423  
“\_debug\_strnset - Set Characters in String” on page 425  
“\_debug\_strset - Set Characters in String” on page 426  
“\_debug\_uheapmin - Free Unused Memory in User Heap”  
“\_debug\_umalloc - Reserve Memory Blocks from User Heap” on page 431  
“heapdebug” on page 270

---

## **`_debug_uheapmin` - Free Unused Memory in User Heap**

### Format

```
#include <umalloc.h>
int _debug_uheapmin(Heap_t heap, const char *file, size_t line);
```

**Language Level:** Extension

`_debug_uheapmin` is the debug version of `_uheapmin`. Like `_uheapmin`, it returns all unused memory blocks from the specified *heap* to the operating system.

To return the memory, `_debug_uheapmin` calls the *release\_fn* you supplied when you created the heap with `_ucreate`. If you do not supply a *release\_fn*, `_debug_uheapmin` has no effect and returns 0.

In addition, `_debug_uheapmin` makes an implicit call to `_uheap_check` to validate the heap.

`_debug_uheapmin` works just like `_debug_heapmin` except that you specify the heap to use; `_debug_heapmin` always uses the default heap.

To use `_debug_uheapmin`, you must compile with the debug memory `-qheapdebug` compiler option. This option maps all `_uheapmin` calls to `_debug_uheapmin`.

**Note:** The `-qheapdebug` option maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

### Return Value

If successful, `_debug_uheapmin` returns 0. A nonzero return value indicates failure. If the heap specified is not valid, `_debug_uheapmin` generates an error message with the file name and line number where the call to `_debug_uheapmin` was made.

## Example

This example creates a heap and allocates memory from it, then uses `_debug_heapmin` to release the memory.

**Note:** You must compile this example with the `-qheapdebug` option to map the `_uheapmin` calls to `_debug_uheapmin`.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <umalloc.h>

int main(void)
{
    Heap_t myheap;
    char *ptr;

    /* Use default heap as user heap */
    myheap = _udefault(NULL);

    /* Allocate a large object */
    if (NULL == (ptr = (char*)_umalloc(myheap, 60000))) {
        puts("Cannot allocate memory from user heap.\n");
        exit(EXIT_FAILURE);
    }
    memset(ptr, 'x', 60000);
    free(ptr);

    /* _debug_uheapmin will attempt to return the freed object to the system */
    if (0 != _uheapmin(myheap)) {
        puts("_debug_uheapmin returns failed.\n");
        exit(EXIT_FAILURE);
    }
    return 0;
}
```

## Related Reading

- “Debugging Memory Heaps” on page 183
- “Memory Management Functions” on page 179
- “Managing Memory with Multiple Heaps” on page 182
- “Debugging Programs with Heap Memory” on page 195
- “heapdebug” on page 270
- “\_debug\_calloc - Allocate and Initialize Memory” on page 407
- “\_debug\_free - Free Allocated Memory” on page 408
- “\_debug\_heapmin - Free Unused Memory in the Default Heap” on page 410
- “\_debug\_malloc - Allocate Memory” on page 412
- “\_debug\_memcpy - Copy Bytes” on page 413
- “\_debug\_memmove - Copy Bytes” on page 415
- “\_debug\_memset - Set Bytes to Value” on page 416
- “\_debug\_realloc - Reallocate Memory Block” on page 417
- “\_debug\_strcat - Concatenate Strings” on page 419
- “\_debug\_strcpy - Copy Strings” on page 421
- “\_debug\_strncat - Concatenate Strings” on page 422
- “\_debug\_strncpy - Copy Strings” on page 423
- “\_debug\_strnset - Set Characters in String” on page 425
- “\_debug\_strset - Set Characters in String” on page 426
- “\_debug\_ucalloc - Reserve and Initialize Memory from User Heap” on page 428
- “\_debug\_umalloc - Reserve Memory Blocks from User Heap”
- “heapdebug” on page 270

---

## `_debug_umalloc` - Reserve Memory Blocks from User Heap

### Format

```
#include <umalloc.h>
void *_debug_umalloc(Heap_t heap, size_t size,
                    const char *file, size_t line);
```

## Language Level: Extension

**\_debug\_umalloc** is the debug version of **\_umalloc**. Like **\_umalloc**, it reserves storage space from the *heap* you specify for a block of *size* bytes. **\_debug\_umalloc** also sets all the memory it allocates to 0xAA, so you can easily locate instances where your program uses the data in the memory without initializing it first.

In addition, **\_debug\_umalloc** makes an implicit call to **\_uheap\_check**, and stores the name of the file *file* and the line number *line* where the storage is allocated. This information can be used later by the **\_uheap\_check**, **\_udump\_allocated**, or **\_udump\_allocated\_delta** functions. **\_debug\_umalloc** also sets all the memory it allocates to 0xAA; this can help you debug problems where your program uses the data in the memory without initializing it.

**\_debug\_umalloc** works just like **\_debug\_malloc** except that you specify the heap to use; **\_debug\_malloc** always allocates from the default heap.

If the *heap* does not have enough memory for the request, **\_debug\_umalloc** calls the *getmore\_fn* that you specified when you created the heap with **\_ucreate**.

To use **\_debug\_umalloc**, you must compile with the debug memory **-qheapdebug** compiler option. This option maps all **\_umalloc** calls to **\_debug\_umalloc**.

**Note:** The **-qheapdebug** option maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

To reallocate or free memory allocated with **\_debug\_umalloc**, use the non-heap-specific **\_debug\_realloc** and **\_debug\_free**. These functions always check what heap the memory was allocated from.

## Return Value

**\_debug\_umalloc** returns a pointer to the reserved space. If *size* was specified as zero, or the *getmore\_fn* cannot provide enough memory, **\_debug\_umalloc** returns NULL. Passing **\_debug\_umalloc** a heap that is not valid results in undefined behavior.

## Example

This example creates a heap *myheap* and uses **\_debug\_umalloc** to allocate 100 bytes from it. It then attempts to overwrite storage that was not allocated. The call to **\_debug\_free** invokes **\_uheap\_check**, which detects the error, generates messages, and ends the program.

**Note:** You must compile this example with the **-qheapdebug** option to map **\_umalloc** to **\_debug\_umalloc**, and **free** to **\_debug\_free**.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>
#include <string.h>

int main(void)
{
    Heap_t myheap;
    char *ptr;

    /* Use default heap as user heap */
    myheap = _udefault(NULL);
```

```

if (NULL == (ptr = (char*)_umalloc(myheap, 100))) {
    puts("Cannot allocate memory from user heap.\n");
    exit(EXIT_FAILURE);
}
memset(ptr, 'x', 105); /* Overwrites storage that was not allocated */
free(ptr);
return 0;

/*****
The output should be similar to :

End of allocated object 0x00073890 was overwritten at 0x000738f4.
The first eight bytes of the memory block (in hex) are: 7878787878787878.
This memory block was (re)allocated at line number 14 in _debug_umallo.c.
Heap state was valid at line 14 of _debug_umallo.c.
Memory error detected at line 19 of _debug_umallo.c.
*****/
}

```

### Related Reading

- “Debugging Memory Heaps” on page 183
- “Memory Management Functions” on page 179
- “Managing Memory with Multiple Heaps” on page 182
- “Debugging Programs with Heap Memory” on page 195
- “heapdebug” on page 270
- “\_debug\_calloc - Allocate and Initialize Memory” on page 407
- “\_debug\_free - Free Allocated Memory” on page 408
- “\_debug\_heapmin - Free Unused Memory in the Default Heap” on page 410
- “\_debug\_malloc - Allocate Memory” on page 412
- “\_debug\_memcpy - Copy Bytes” on page 413
- “\_debug\_memmove - Copy Bytes” on page 415
- “\_debug\_memset - Set Bytes to Value” on page 416
- “\_debug\_realloc - Reallocate Memory Block” on page 417
- “\_debug\_strcat - Concatenate Strings” on page 419
- “\_debug\_strcpy - Copy Strings” on page 421
- “\_debug\_strncat - Concatenate Strings” on page 422
- “\_debug\_strncpy - Copy Strings” on page 423
- “\_debug\_strnset - Set Characters in String” on page 425
- “\_debug\_strset - Set Characters in String” on page 426
- “\_debug\_ucalloc - Reserve and Initialize Memory from User Heap” on page 428
- “\_debug\_uheapmin - Free Unused Memory in User Heap” on page 430
- “heapdebug” on page 270



## Appendix G. Built-in Functions for PowerPC Processors

PowerPC platforms support certain RS/6000 machine instructions not available on other platforms. If performance is critical to your application, the C for AIX compiler provides a set of built-in functions that directly map to certain RS/6000 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.

| <u>Name</u>                                           | <u>Prototype</u>                                                                                              | <u>Return Value or Action Performed</u>                                                                                                                                                              |
|-------------------------------------------------------|---------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>__fmadds( )</code><br><code>__fmadd( )</code>   | float <code>__fmadds</code> (float, float, float);<br>double <code>__fmadd</code> (double, double, double);   | <code>__fmadds</code> (a, x, y) = [a * x + y]<br><code>__fmadd</code> (a, x, y) = [a * x + y]                                                                                                        |
| <code>__fmsubs( )</code><br><code>__fmsub( )</code>   | float <code>__fmsubs</code> (float, float, float);<br>double <code>__fmsub</code> (double, double, double);   | <code>__fmsubs</code> (a, x, y) = [a * x - y]<br><code>__fmsub</code> (a, x, y) = [a * x - y]                                                                                                        |
| <code>__fnmadds( )</code><br><code>__fnmadd( )</code> | float <code>__fnmadds</code> (float, float, float);<br>double <code>__fnmadd</code> (double, double, double); | <code>__fnmadds</code> (a, x, y) = [- (a * x + y)]<br><code>__fnmadd</code> (a, x, y) = [- (a * x + y)]                                                                                              |
| <code>__fnmsubs( )</code><br><code>__fnmsub( )</code> | float <code>__fnmsubs</code> (float, float, float);<br>double <code>__fnmsub</code> (double double, double);  | <code>__fnmsubs</code> (a, x, y) = [- (a * x - y)]<br><code>__fnmsub</code> (a, x, y) = [- (a * x - y)]                                                                                              |
| <code>__fsqrts( )</code><br><code>__fsqrt( )</code>   | float <code>__fsqrts</code> (float);<br>double <code>__fsqrt</code> (double);                                 | <code>__fsqrts</code> (x) = square root of x<br><code>__fsqrt</code> (x) = square root of x                                                                                                          |
| <code>__frsqrte( )</code>                             | double <code>__frsqrte</code> (double);                                                                       | <code>__frsqrte</code> (x) = [(estimate of) 1.0/sqrt(x)]                                                                                                                                             |
| <code>__fres( )</code>                                | float <code>__fres</code> (float);                                                                            | <code>__fres</code> (x) = [(estimate of) 1.0/x]                                                                                                                                                      |
| <code>__fsels( )</code><br><code>__fsel( )</code>     | float <code>__fsels</code> (float, float, float);<br>double <code>__fsel</code> (double, double, double);     | if (a >= 0.0) then <code>__fsels</code> (a, x, y) = x;<br>else <code>__fsels</code> (a, x, y) = y<br>if (a >= 0.0) then <code>__fsel</code> (a, x, y) = x;<br>else <code>__fsel</code> (a, x, y) = y |
| <code>__fabss( )</code><br><code>__fabs( )</code>     | float <code>__fabss</code> (float);<br>double <code>__fabs</code> (double);                                   | <code>__fabss</code> (x) =  x <br><code>__fabs</code> (x) =  x                                                                                                                                       |
| <code>__fnabss( )</code><br><code>__fnabs( )</code>   | float <code>__fnabss</code> (float);<br>double <code>__fnabs</code> (double);                                 | <code>__fnabss</code> (x) = - x <br><code>__fnabs</code> (x) = - x                                                                                                                                   |
| <code>__dcbt( )</code>                                | void <code>__dcbt</code> (void *);                                                                            | Data Cache Block Touch. Loads the block of memory containing the specified address into the data cache.                                                                                              |
| <code>__dcbz( )</code>                                | void <code>__dcbz</code> (void *);                                                                            | Data Cache Block set to Zero. Sets the specified address in the data cache to zero (0).                                                                                                              |
| <code>__trap( )</code>                                | void <code>__trap</code> (int);                                                                               | Trap if the parameter is not zero.                                                                                                                                                                   |

| <u>Name</u>             | <u>Prototype</u>                       | <u>Return Value or Action Performed</u> |
|-------------------------|----------------------------------------|-----------------------------------------|
| <code>__trapd( )</code> | <code>void __trapd (long long);</code> | Trap if the parameter is not zero.      |

**Related Reading**

“Acceptable Compiler Mode and Processor Architecture Combinations” on page 16



## Appendix H. RISC System/6000 Alignment Rules

On the RISC System/6000 system, an aggregate is aligned according to its most strictly aligned member. Within aggregates, members are aligned according to their type. The table below summarizes size and alignment information for each type.

| Type, Size, and Alignment for the RISC System/6000 System        |                                                                              |              |
|------------------------------------------------------------------|------------------------------------------------------------------------------|--------------|
| Type                                                             | Alignment of Member                                                          | Size (bytes) |
| char                                                             | byte aligned                                                                 | 1            |
| short                                                            | halfword aligned                                                             | 2            |
| (long) int                                                       | word aligned                                                                 | 4            |
| long long int                                                    | doubleword aligned                                                           | 8            |
| pointer                                                          | word aligned                                                                 | 4            |
| float                                                            | word aligned                                                                 | 4            |
| double                                                           | doubleword aligned if <b>-qalign=natural</b> . Otherwise, word aligned.      | 8            |
| long double with <b>-qlongdouble</b> or <b>-qldbl128</b> option. | long doubleword aligned if <b>-qalign=natural</b> . Otherwise, word aligned. | 16           |

### Notes:

1. The entire object is aligned on the same boundary as its most strictly aligned member.
2. Each member is assigned the lowest available offset with the appropriate alignment (internal padding).
3. The object's size is increased, if necessary, to make it a multiple of the size of its most strictly aligned member. (For example, if the object contains a word, it is padded to a word boundary.)

On the RISC System/6000 system, if a **double** is the first member of a **struct**, it is 8-byte (doubleword) aligned. If a **long double** is the first member of a **struct**, it is 16-byte aligned.

### Bit Fields for RISC System/6000 Format

The following rules apply when you are laying out bit fields in **structs**.

- **structs** containing bit fields are 4-byte (word) aligned.
- Bit fields can be at most 32 bits long.
- Bit fields are packed into the current word. If a bit field would cross a word boundary, it starts at the next word boundary.
- A bit field of width zero causes the bit field that immediately follows it to be aligned at the next word boundary. If the zero width bit field is at a word boundary, the next bit field starts at this boundary.
- A **struct** containing nothing but zero-width bit fields is allowed and will have a length of 0 bytes.

In the C language, you can specify bit fields as **char** or **short** instead of **int**, but the compiler maps them as if they were **unsigned int**. In **extended** mode, you can use the **sizeof** operator on a bit field. (The **sizeof** operator on a bit field always returns 4.)

### Related Reading

“align” on page 234  
“\_\_align Specifier” on page 442  
“MacIntosh and Twobyte Alignment Rules” on page 440  
“Packed Alignment Rules”  
“Alignment Rules for Nested Aggregates”

---

## Alignment Rules for Nested Aggregates

Aggregates with different alignments can be nested. Each aggregate is laid out using the alignment rules applicable to it. The start position of the nested aggregate is determined by the alignment rules of the aggregate in which it is nested.

### Example

The following example uses these symbols to show padding and boundaries:

```
p = padding
| = halfword boundary
: = byte boundary
```

For:

```
#pragma options align=mac68k
struct A {
    char a;
    #pragma options align=power
    struct B {
        int b;
        char c;
    } B1; // ← B1 laid out using RISC System/6000 alignment rules
    #pragma options align=reset // ← has no effect on A or B, but
                                // on subsequent structs
    char d;
};
sizeof(A) == 12
```

The layout of A is:

```
|a:p|b:b|b:b|c:p|p:p|d:p|
```

### Related Reading

“align” on page 234  
“\_\_align Specifier” on page 442  
“Appendix H. RISC System/6000 Alignment Rules” on page 437  
“MacIntosh and Twobyte Alignment Rules” on page 440  
“Packed Alignment Rules”

---

## Packed Alignment Rules

All structures are byte-aligned regardless of their members. All members are also byte-aligned. (Bit fields are byte-aligned, but bit-field members are not.)

## Example

The following example uses these symbols to show padding and boundaries:

```
p = padding
| = halfword boundary
: = byte boundary
```

For:

```
#pragma options align=packed
struct {
    char a;
    double b;
} B;
#pragma options align=reset
sizeof(B) == 9
```

The layout of B is:

```
|a:b|b:b|b:b|b:
```

## Packed Bit Fields

The following rules apply when laying out packed bit fields.

- An individual bit field can be at most 32 bits long.
- Bit fields are packed together into the current word. If a bit field extends beyond the current word, it starts at the next byte boundary.
- A bit field of width zero causes the next class member to start at the next byte boundary. If the zero-width bit field is already at a byte boundary, the next structure member starts at this boundary.
- A nonbit field following a bit field is aligned on the next byte boundary.

## Example

```
#pragma options align=packed
struct {
    int a : 8;
    int b : 10;
    int c : 12;
    int d : 4;
    int e : 3;
    int : 0;
    int f : 1;
    char g;
} A;
#pragma options align=reset
sizeof(A) == 7
```

The layout of A is:

| Member Name | Displacement<br>bytes (bits) |
|-------------|------------------------------|
| a           | 0                            |
| b           | 1                            |
| c           | 2 (2)                        |
| d           | 4                            |
| e           | 4 (4)                        |

f 5  
g 6

### Related Reading

“align” on page 234  
 “Appendix H. RISC System/6000 Alignment Rules” on page 437  
 “MacIntosh and Twobyte Alignment Rules”  
 “Alignment Rules for Nested Aggregates” on page 438

---

## MacIntosh and Twobyte Alignment Rules

All unions and structures are halfword aligned regardless of their members. Within the aggregate, members are aligned according to their type. The size of types for the Macintosh system is the same as on the RISC System/6000 system. The table below summarizes alignment information for each type.

| Type, Size, and Alignment for the Macintosh System                     |                  |               |
|------------------------------------------------------------------------|------------------|---------------|
| Type                                                                   | Alignment        | Size          |
| char                                                                   | byte aligned     | byte          |
| short                                                                  | halfword aligned | halfword      |
| (long) int                                                             | halfword aligned | word          |
| long long int                                                          | halfword aligned | doubleword    |
| pointer                                                                | halfword aligned | word          |
| float                                                                  | halfword aligned | word          |
| double                                                                 | halfword aligned | doubleword    |
| long double                                                            | halfword aligned | doubleword    |
| long double<br>with <b>-qlongdouble</b><br>or <b>-qldbl128</b> option. | halfword aligned | quadrupleword |

### Example

The following example uses these symbols to show padding and boundaries:

```
p = padding
| = halfword boundary
: = byte boundary
```

For:

```
#pragma options align=mac68k
struct A {
    char a;
}
sizeof(A) == 2
```

The layout of A is:

```
|a:p|
```

For:

```
#pragma options align=mac68k
struct B {
    char a;
    double b;
}
sizeof(B) == 10
```

The layout of B is:

```
|a:p|b:b|b:b|b:b|b:b|
```

### Bit fields for Macintosh Format

The following rules apply when you are laying out bit fields in structures.

- An individual bit field can be at most 32 bits long.
- Bit fields are packed into a word and are aligned on a 2-byte boundary.
- Bit fields that would cross a word boundary are moved to the *next* halfword boundary even if they are already starting on a halfword boundary. (The bit field may still end up crossing a word boundary.)
- A bit field of width zero forces the next member (even if it is not a bit field) to start at the *next* halfword boundary even if the zero-width bit field is currently at a halfword boundary.
- A structure containing nothing but zero width bit fields is allowed and will have a length, in bytes, of two times the number of zero width bit fields.

For unions, there is one special case:

- Unions whose largest element is a bit field of width 16 or less have a size of 2 bytes. If the width of the bit field is greater than 16, the size of the union is 4 bytes.

### Example

The following example uses these symbols to show padding and boundaries:

```
p = padding
| = halfword boundary
: = byte boundary
```

For:

```
#pragma options align=mac68k
struct A {
    char a;
    int : 0;
    int b : 4;
    int c : 17;
}
sizeof(A) == 8
```

The layout of A is:

```
|a:p|b .. :p|c:c|c .. :p|
```

### Type Compatibility between RISC System/6000 and Macintosh Systems

Different aggregate types with identical members are not compatible. Therefore such aggregates cannot be assigned to each other. Use the **-qextchk** option to assure type safe linkage for function declarations that may use different alignment rules.

## Related Reading

“align” on page 234

“Appendix H. RISC System/6000 Alignment Rules” on page 437

“Packed Alignment Rules” on page 438

“Alignment Rules for Nested Aggregates” on page 438

“extchk” on page 258

---

## \_\_align Specifier

### Syntax

```
declarator __align (integer_constant) identifier;  
struct_or_union_specifier __align (integer_constant) [identifier {struct_declaration_list}];
```

where:

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

### Purpose

Use the **\_\_align** specifier to explicitly specify alignment and padding when declaring or defining data items.

### 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 within an aggregate 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.

### 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 Reading

- “align” on page 234
- “Appendix H. RISC System/6000 Alignment Rules” on page 437
- “MacIntosh and Twobyte Alignment Rules” on page 440
- “Alignment Rules for Nested Aggregates” on page 438





---

## Appendix I. Implementation Dependencies Overview

The behavior of some features of the C language depends on the particular C compiler used. This and related pages describe these C for AIX implementation-specific dependencies.

- “C for AIX Compiler Limits”
- “Implementation-Defined Behavior”
- “Type Conversions” on page 461
- “Synchronization of Stores and Loads to I/O Space” on page 464

### Note: Compiling your C for AIX programs with C for AIX

The C for AIX compiler is a stricter compiler than the XL C compiler. See “Appendix J. C for AIX and XL C Compatibility” on page 467, which discusses the issues you should consider if you plan to use your XL C programs with the C for AIX compiler.

---

## C for AIX Compiler Limits

The limits for your program might differ from those stated here depending on its complexity.

| Language Feature                              | Limit                                                                        |
|-----------------------------------------------|------------------------------------------------------------------------------|
| Nesting levels for included files             | 255                                                                          |
| Significant initial characters in identifiers | No limit (but the linker has a limit of 4095 characters for external names). |

Other system limits are set in the `/usr/include/sys/limits.h` file. The `limits.h` header file is described in the “Header Files Overview” in the *AIX Version 4 Files Reference*.

### Related Reading

“Appendix I. Implementation Dependencies Overview”

---

## Implementation-Defined Behavior

Appendix F and other sections of the *American National Standard for Information Systems — Programming Language C (X3.159-1989)* standard designate several C language features as *implementation-defined*. The ANSI standard for the C language requires that manufacturers document the specifics of these features for their compiler.

This and related pages provide this ANSI-required documentation, and references the relevant sections of the ANSI standard. The references are given in parentheses for each item, for example, **(2.1.1.2)** .

- “Implementation Dependency - Translation (F.3.1)” on page 446
- “Implementation Dependency - Environment (F.3.2)” on page 447
- “Implementation Dependency - Identifiers (F.3.3)” on page 448
- “Implementation Dependency - Characters (F.3.4)” on page 448
- “Implementation Dependency - Integers (F.3.5)” on page 449
- “Implementation Dependency - Floating Point Types (F.3.6)” on page 451
- “Implementation Dependency - Arrays and Pointers (F.3.7)” on page 452
- “Implementation Dependency - Registers (F.3.8)” on page 452

- “Implementation Dependency - Structures, Unions, Enumerations, Bit Fields (A.6.3.9) (F.3.9)” on page 453
- “Implementation Dependency - Qualifiers (F.3.10)” on page 453
- “Implementation Dependency - Declarators (F.3.11)” on page 454
- “Implementation Dependency - Statements (F.3.12)” on page 454
- “Implementation Dependency - Preprocessing Directives (F.3.13)” on page 454
- “Implementation Dependency - Library Functions (F.3.14)” on page 455
- “Implementation Dependency - Locale-Specific Behavior (F.4)” on page 459

### Related Reading

“Appendix I. Implementation Dependencies Overview” on page 445

## Implementation Dependency - Translation (F.3.1)

*Translation* is the process of transforming all or part of a source program into a program image that contains all the information needed for the program to run.

Each nonempty sequence of white-space characters in a source program, other than the new-line character, is replaced by one space character. Comments are treated as white space. **(2.1.1.2)**

If you use the **-C** compiler option, comments are retained.

Diagnostic messages have the following format **(2.1.1.3)**:

*15cc-nnn (severity) text.*

where:

|                 |                                                                                |
|-----------------|--------------------------------------------------------------------------------|
| <i>cc</i>       | Is a two-digit code indicating the compiler component that issued the message: |
|                 | <b>00</b> Code-generation or optimization message                              |
|                 | <b>01</b> Compiler services message                                            |
|                 | <b>05</b> Front-end text message                                               |
|                 | <b>06</b> Front-end error message                                              |
| <i>nnn</i>      | Is the message number                                                          |
| <i>severity</i> | Is a letter representing the severity level of the message                     |
| <i>text</i>     | Is the message text describing the error                                       |

Diagnostic messages have the following severity levels:

| 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. |

The translator returns the following status codes for each severity level of diagnostic message:

| Message Severity                 | Return Code |
|----------------------------------|-------------|
| Informational, Warning, or Error | 0           |
| Severe                           | 1           |
| Unrecoverable                    | 40          |

The level of diagnostic can be controlled with the following compiler options (2.1.1.3):

```
halt=severity
flag=severity1:severity2
```

See “Message Severity Levels and Compiler Response” on page 20 for detailed information about C for AIX compiler diagnostic messages.

#### Related Reading

- “Implementation-Defined Behavior” on page 445
- “Appendix I. Implementation Dependencies Overview” on page 445
- “Message Severity Levels and Compiler Response” on page 20
- “Compiler Return Codes” on page 20
- “C” on page 242
- “flag” on page 261
- “halt” on page 269

## Implementation Dependency - Environment (F.3.2)

Program termination and library facilities are *not* available to a free-standing program. Only a hosted environment is available. (2.1.2.1)

Arguments passed to a C program from the AIX operating system command line can be retrieved as arguments of the function **main**. (2.1.2.2.1) The first argument passed indicates the number of white-space-separated parameters supplied. The second argument is an array of pointers to these parameters. The third argument is a null-terminated array of pointers to a set of system arguments for the program. The arguments are traditionally named **argc**, **envp[]**, and **argv[]**, but these names are not mandatory. To access these variables, the **main()** function of your program must declare them as arguments. The following table summarizes the **main()** function arguments and their uses.

| Name     | Meaning        | Purpose                                                                                                                             |
|----------|----------------|-------------------------------------------------------------------------------------------------------------------------------------|
| int argc | Argument count | Contains the number of parameters supplied on the command line. The name with which the program was invoked is the first parameter. |

|                      |                     |                                                                                                                                                                                                                                                                           |
|----------------------|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>char*</b> argv[ ] | Argument vector     | An array of pointers to character strings. Each character string is a parameter supplied on the command line. Valid pointers are from argv[0] to argv[argc-1]. argv[0] is the command name, and argv[1] contains what was specified as the first argument to the command. |
| <b>char*</b> envp[ ] | Environment pointer | An array of pointers to character strings. Each character string is an environment parameter supplied to the program. The array ends at the first null pointer (array element with a value of zero).                                                                      |

An asynchronous terminal and a paired display and keyboard are the valid interactive devices. **(2.1.2.3)**

#### Related Reading

“Implementation-Defined Behavior” on page 445

“Appendix I. Implementation Dependencies Overview” on page 445

## Implementation Dependency - Identifiers (F.3.3)

In an identifier without external linkage, 219 characters beyond 31 (for a total of 250) are significant. **(3.1.2)**

In an identifier with external linkage, 244 characters beyond 6 (for a total of 250) are significant. **(3.1.2)**

Case is significant in an identifier with external linkage.

#### Related Reading

“Implementation-Defined Behavior” on page 445

“Appendix I. Implementation Dependencies Overview” on page 445

## Implementation Dependency - Characters (F.3.4)

The C for AIX compiler uses all source and execution characters explicitly specified by the ANSI C Standard. **(2.2.1)**

The C for AIX compiler recognizes the following escape sequence values **(2.2.2)**:

| Sequence | Decimal Value | Character Represented |
|----------|---------------|-----------------------|
| \a       | 7             | Alert (Bell)          |
| \b       | 8             | Backspace             |
| \f       | 12            | Form feed             |
| \n       | 10            | New line              |
| \r       | 13            | Carriage return       |
| \t       | 9             | Horizontal tab        |
| \v       | 11            | Vertical tab          |
| \"       | 34            | Double quotation mark |
| \'       | 39            | Single quotation mark |
| \?       | 63            | Question mark         |

|    |    |           |
|----|----|-----------|
| \\ | 92 | Backslash |
|----|----|-----------|

Multibyte characters are encoded such that the first character identifies the form of the multibyte character. Any character is valid in the remaining bytes. **(2.2.1.2)**

A character in the execution character set contains 8 bits. **(2.2.4.2)**

The mapping of members of the source character set (in character and string literals) to members of the execution character set is one to one. **(3.1.3.4)**

The same code set is used for both source and execution set. **(3.1.3.4)**

The lowest 4 bytes represent character constants when **(3.1.3.4)**:

- An integer-character constant contains a character or escape sequence not represented in the basic execution character set.
- A wide-character constant contains a character or escape sequence not represented in the extended character set.
- An integer character constant contains more than one character.
- A wide-character constant contains more than one multibyte character.

The name of the locale used to convert multibyte characters into corresponding wide characters (codes) for a wide-character constant is user-defined. **(3.1.3.4)**

The default equivalent type of a plain **char** is **unsigned char**. **(3.2.1.1)** The **chars** option sets the sign type of characters.

#### Related Reading

“Implementation-Defined Behavior” on page 445

“Appendix I. Implementation Dependencies Overview” on page 445

“chars” on page 244

## Implementation Dependency - Integers (F.3.5)

The following table shows the storage occupied and the range of various integer types. **(3.1.2.5)**

| Type               | Size (bits) | Range (in limits.h)                                 |
|--------------------|-------------|-----------------------------------------------------|
| signed char        | 8           | $-(127 + 1)$ to 127                                 |
| (unsigned) char    | 8           | 0 to 255                                            |
| (signed) short     | 16          | $-(32767 + 1)$ to 32767                             |
| unsigned short     | 16          | 0 to 65535                                          |
| (signed) int       | 32          | $-(2147483647 + 1)$ to 2147483647                   |
| unsigned int       | 32          | 0 to 4294967295                                     |
| (signed) long      | 32          | $-(2147483647 + 1)$ to 2147483647                   |
| unsigned long      | 32          | 0 to 4294967295                                     |
| (signed) long long | 64          | $-(9223372036854775807 + 1)$ to 9223372036854775807 |
| unsigned long long | 64          | 0 to 18446744073709551615                           |

When an integer is converted to a **signed char**, the lowest byte of the integer is used to represent the **char**. **(3.2.1.2)**

When an integer is converted to a shorter **signed** integer, the lowest 2 bytes of the integer are used to represent the **short int**. (3.2.1.2)

When an **unsigned** integer is converted to a **signed** integer of equal length, the bit pattern is preserved, and the high-order bit becomes the sign bit. (3.2.1.2)

Bitwise operations (Inclusive **OR** (`|`), **AND** (`&`), Exclusive **OR** (`^`)) on a **signed** integer use the bit pattern of the signed quantity. (3.3)

If either operand of integer division is negative, the result of the `/` operator is the largest integer less than or equal to the algebraic quotient. (3.3.5)

When a bitwise right shift (`>>`) of a negative **signed** integral type is used, vacated bits are filled with ones. (3.3.7) When a bitwise right shift (`>>`) of a non-negative **signed** integral type or an **unsigned** integral type is used, vacated bits are filled with zeros.

When using **long long** in your programs, the following rules apply:

- The size of pointers remains 32-bit.
- An integer constant has type **long long int** only if it is suffixed by the letters **ll**.
- An integer constant with suffix **ll** has type **long long int** or **unsigned long long int**. An integer with suffix **ull** or **llu** has type **unsigned long long int**.
- Integral promotions are unchanged. Integral promotions are not performed on **long long ints**.
- Conversion of an integral type to **long long int** only occurs if one of the operands in the expression being evaluated has type **long long int**.
- When a value of **float** type is converted to **long long** type, the fractional part is discarded. If the value of the integral part cannot be represented by the **long long int** type, the result is the maximum value of the **long long int** type.
- When a value of **long long int** type is converted to floating type, if the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower value, depending on the IEEE rounding mode. You set the compile-time IEEE rounding mode with the `-y` compiler option.
- For binary operators that expect operands of arithmetic type, if either operand has type **unsigned long long int**, the other operand is converted to **unsigned long long int**. Otherwise, if either operand has type **long long int**, the other operand is converted to **long long int**.
- The maximum size of a bit-field remains 32-bit.
- A conversion specification may contain an optional **ll**, specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **long long int** or **unsigned long long int** argument.
- For **power** alignment rules, **long long int** identifiers and aggregates containing **long long int** members are aligned on a 64-bit boundary.
- For type-based aliasing, lvalues of type **char**, **unsigned char**, **long long int**, or **unsigned long long int** are considered as possible aliases of a **long long int** or **unsigned long long int**.
- The attribute listing will display long long integer for **signed long long int** identifiers and unsigned long long integer for **long long int** identifiers.
- In interlanguage calls, a C **signed long long int** maps to a Fortran **INTEGER\*8** and an **unsigned long long int** maps to a Fortran **LOGICAL\*8**.

Other system limits are set in the `/usr/include/sys/limits.h` file. The `limits.h` header file is described in the Header Files Overview of the *AIX Version 4 Files Reference*.

## Related Reading

“Implementation-Defined Behavior” on page 445

“Appendix I. Implementation Dependencies Overview” on page 445

“Type Conversions” on page 461

## Implementation Dependency - Floating Point Types (F.3.6)

The scaled value of a floating-point constant that is in the range of the representable value for its type is controlled at compile time by the **-y** compiler option, and conforms to the IEEE standard for binary floating-point arithmetic. **(3.1.3.1)**

The following table shows the storage occupied and the range of various floating-point types **(3.1.2.5)**:

| Type                          | Size (bits) | Range of base 10 Exponents | Range of Decimal Values (in float.h)               | Precision (decimal digits) |
|-------------------------------|-------------|----------------------------|----------------------------------------------------|----------------------------|
| float                         | 32          | -37 to 38                  | 1.175494351E-38 to 3.402823466E+38                 | 7                          |
| double                        | 64          | -307 to 308                | 2.2250738585072014E-308 to 1.7976931348623157E+308 | 15                         |
| long double                   | 64          | -307 to 308                | 2.2250738585072014E-308 to 1.7976931348623157E+308 | 15                         |
| long double (-qdbl128 option) | 128         | -307 to 308                | 2.2250738585072014E-308 to 1.7976931348623157E+308 | 31                         |

Other floating-point limits are set in the `/usr/include/float.h` header file, described in “Header Files Overview” in the *AIX Version 4 Files Reference*.

When an integral value is converted to a floating-point number that cannot exactly represent the original value, the direction of truncation depends on the compile-time rounding mode set by the **-y** compiler option. **(3.2.1.3)**

When a floating-point number is converted to a narrower floating-point number, the direction of truncation or rounding depends on the rounding mode set by the **-y** compiler option. **(3.2.1.4)**

### Using 16-byte long doubles (-qdbl128 Option)

The mathematical functions contained in the `<math.h>` header file, such as `cosl`, `tanl`, `fmodl`, have been updated to work with 16-byte **long double** floating-point numbers.

The input/output functions in the `<stdio.h>` header file, such as `printf`, `scanf`, `vsprintf`, have been updated to work with 16-byte **long double** floating-point numbers.

A new function `atold`, which converts a string to a **long double** representation, has been added to the existing string-to-number functions `strtod`, `strtoul`, and `strtoul`.

### Alignment Rules

If the first member of a union or structure is a **long double**, the aggregate is aligned on a 128-bit boundary. Other aggregates and **long double** identifiers are aligned on a 32-bit boundary. If `-qalign=natural` is specified, all **long doubles** are aligned on a 128-bit boundary, regardless of their placement in a union or structure.

The 2-byte alignment rules remain unchanged. All identifiers and aggregates are aligned on a 16-bit boundary.

For bind-time type checking, a **long double** has type **r16**.

There are three floating types: **float**, **double**, and **long double**. The range of values of each type is a subrange of the values of the next type in the list.

When the compiler converts a value of floating type to integral type, the fractional part is discarded. If the value of the integral part is too large to be represented by an integral type, the value is converted to the maximum value of an integral type.

When a **long double** is demoted to **double** or **float**, if the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower value, depending on the IEEE rounding mode.

#### **Related Reading**

“Implementation-Defined Behavior” on page 445

“Appendix I. Implementation Dependencies Overview” on page 445

“y” on page 339

“align” on page 234

“ldbl128, longdouble” on page 289

## **Implementation Dependency - Arrays and Pointers (F.3.7)**

The type of the integer required to hold the maximum size of an array (the type of the **sizeof** operator, **size\_t**) is **unsigned long**. **(3.3.3.4 and 4.1.1)**

For a pointer to be converted to an integral type, the size of integer required is 32 or 64 bits. **(3.3.4)**

The type of the integer required to hold the difference between two pointers to elements of the same array (**ptrdiff\_t**) is **long**. **(3.3.6 and 4.1.1)**

When a pointer is cast to an integer, or an integer is cast to a pointer, the bit patterns are preserved. **(3.3.4)**

#### **Related Reading**

“Implementation-Defined Behavior” on page 445

“Appendix I. Implementation Dependencies Overview” on page 445

## **Implementation Dependency - Registers (F.3.8)**

Objects in registers declared with the storage class specifier **register** are treated as **int** objects. **(3.5.1)**

#### **Related Reading**

“Implementation-Defined Behavior” on page 445

“Appendix I. Implementation Dependencies Overview” on page 445

“register” on page 111

“int, long, short” on page 81



## Implementation Dependency - Structures, Unions, Enumerations, Bit Fields (A.6.3.9) (F.3.9)

If a union object is accessed using a member of a different type, the result is undefined. **(3.3.2.3)**

The alignment and padding of structures and unions are subject to the following rules **(3.5.2.1)**:

- Padding is added to align the structure and union members on their natural boundaries.
- Padding is added to the end of structures and unions on their natural boundary.
- The alignment of a structure or union is that of its strictest member.

An **unsigned int** bit field is equivalent to a plain **int** bit field. **(3.5.2.1)**

Bit fields within an **int** are allocated from low memory to high memory. For example, 0x12345678 is stored with byte 0 containing 0x12, and byte 3 containing 0x78. **(3.5.2.1)**

The type of an **enum** bit field is the underlying type of the enumeration.

The maximum bit field length for the compiler is 32 bits. If a series of bit fields does not add up to the size of an **int**, padding may take place.

Bit fields cannot cross a storage-unit boundary. For example, in the following **struct**, with bit fields mapped as shown,

```
struct S {
    unsigned int bit : 30;
    unsigned int bit : 3;
    unsigned int bit : 8;
}
```

the 3-bit bit field following the 30-bit bit field would cross a 32-bit word-storage boundary. Because crossing a word boundary is not allowed, the 30 bit field is padded and the 3-bit field starts another word.

The **-qalign=** option allows you to specify different bit-field alignment rules.

Values of an enumeration type are represented by the integral type **signed int**. **(3.5.2.2)**

You can use the **-qenum=small** compiler option to minimize the size of enumerations to 1, 2, or 4 bytes.

### Related Reading

"Implementation-Defined Behavior" on page 445

"Appendix I. Implementation Dependencies Overview" on page 445

"align" on page 234

"enum" on page 255

## Implementation Dependency - Qualifiers (F.3.10)

For access to an object that has a type qualified as **volatile**, all access is retained. **(3.5.5.3)** The **volatile** attribute maintains consistency in memory access to data objects. Volatile objects are read from memory each time their value is needed, and written back to memory each time they are changed. When optimizing the code, the compiler cannot remove any access to a **volatile** variable. The **volatile** attribute is useful for objects (such as the system clock) that have values that may be changed in ways unknown to your program.

For example, in the following program fragment,

```
volatile int i;  
i=3;  
i=5;  
i=6;
```

all of these statements are kept by the compiler even if the **-O** option to optimize the code is active. If `int i` was not defined as **volatile**, only `i=6` would be kept in the optimized code.

#### Related Reading

"Implementation-Defined Behavior" on page 445  
"Appendix I. Implementation Dependencies Overview" on page 445  
"O, optimize" on page 302 Compiler Option

## Implementation Dependency - Declarators (F.3.11)

There is no limit to the number of declarators that can modify a structure, union, or arithmetic type. **(3.5.4)**

#### Related Reading

"Implementation-Defined Behavior" on page 445  
"Appendix I. Implementation Dependencies Overview" on page 445

## Implementation Dependency - Statements (F.3.12)

There is no limit to the number of **case** values in a **switch** statement. **(3.6.4.2)**

#### RELATED REFERENCES

"Implementation-Defined Behavior" on page 445  
"Appendix I. Implementation Dependencies Overview" on page 445

#### Related Reading

CTR

## Implementation Dependency - Preprocessing Directives (F.3.13)

The value of a single-character character constant in a constant expression that controls conditional inclusion does not match the value of the same character constant in the execution character set. **(3.8.1)**

A single-character character constant in a constant expression that controls conditional inclusion can have a negative value. **(3.8.1)**

The method of locating included source files is described in "Directory Search Sequence for Include Files Using Relative Path Names" on page 178. **(3.8.2)**

Headers are identified by the directive **#include <file\_name>**, and are stored under **/usr/include**. **(3.8.2)**

User-defined include-file names can be specified in double quotation marks (**"**). **(3.8.2)**

The mapping between the name specified in the include directive and the external source file name is one to one. Multibyte characters are permitted in the include directive. **(3.8.2)**

The definitions of the **\_\_DATE\_\_** and **\_\_TIME\_\_** predefined macros are always defined to the system date and time. **(3.8.8)**

## Related Reading

"Implementation-Defined Behavior" on page 445

"Appendix I. Implementation Dependencies Overview" on page 445

"#include Preprocessor Directive" on page 356

## Implementation Dependency - Library Functions (F.3.14)

All of the C library functions are part of the AIX Base Operating System (BOS) Runtime Services and are described in detail in the *AIX Version 4 Technical Reference, Volumes 1 and 2: Base Operating System and Extensions*. Refer to the Subroutines Overview in *AIX Version 4 General Programming Concepts* for general information about library functions.

The macro **NULL**, defined in the `/usr/include/stddef.h` header file, expands to the null pointer constant, **(void \*)0**. (4.1.5)

The **assert** macro, defined in the `/usr/include/assert.h` header file, writes the following message to standard error and stops the program (4.2):

Assertion failed: *expression*, file *file\_name*, line *line\_number*

In the error message, *file\_name* is the name of the source file and *line\_number* is the source line number of the assert statement.

The character-testing (**ctype**) functions (**isalnum**, **isalpha**, **iscntrl**, **islower**, **isprint**, and **isupper**) test for the following characters (4.3.1):

|                |                                                                           |
|----------------|---------------------------------------------------------------------------|
| <b>isalnum</b> | The alphanumeric characters [A-Z], [a-z], and [0-9]                       |
| <b>isalpha</b> | The alphabetic characters [A-Z] and [a-z]                                 |
| <b>iscntrl</b> | The nonprinting characters and any control character (ASCII 0-31, or 127) |
| <b>islower</b> | The lowercase characters [a-z]                                            |
| <b>isprint</b> | The alphanumeric, punctuation, and space printing characters              |
| <b>isupper</b> | The uppercase characters [A-Z]                                            |

After a domain error (**EDOM**), the mathematics functions return the value **NaN**. (4.5.1)

The mathematics functions set the integer expression **errno** to the value of the macro **ERANGE** on underflow range errors. The `/usr/include/sys/errno.h` header file declares the **errno** variable and defines a constant for each of the possible error conditions. (4.5.1)

When the **fmod** function has a second argument of zero, a domain error (**EDOM**) is generated and NaN is returned. (4.5.6.4)

## Signals

The **sigaction**, **sigvec**, and **signal** functions of the C Library specify the action to take upon delivery of a signal. These functions are described in the *AIX Version 4 Technical Reference, Volumes 1 and 2: Base Operating System and Extensions*. (4.7.1.1)

Signals do not have parameters because they are not functions. (4.7.1.1)

The default handling and the handling at program startup for each signal recognized by the **signal** function are described with the **sigaction**, **sigvec**, and **signal** functions. (4.7.1.1)

The equivalent of **signal (sig,SIG\_DFL)** is performed before the call of a signal handler. (4.7.1.1)

The default handling is reset if a **SIGKILL** signal is received by a handler specified to the **signal** function. **(4.7.1.1)**

### **Streams and Files**

The last line of a text stream does not require a terminating new-line character. **(4.9.2)**

Space characters that are written out to a text stream immediately before a new-line character appear when the stream is read back in. **(4.9.2)**

No null characters can be appended to data written to a binary stream. **(4.9.2)**

The file position indicator of an append mode stream is initially positioned at the end of the file. **(4.9.3)**

A write on a text stream does not cause the associated file to be truncated beyond that point. **(4.9.3)**

Default file buffering is fully buffered. File buffering can be set to line or to unbuffered with the **setbuf** function. **(4.9.3)**

A zero-length file can exist. **(4.9.3)**

The **/** character is not allowed in a valid file name. **(4.9.3)**

The same file can be opened multiple times. **(4.9.3)**

The **remove** function does not remove an open file. The file is removed after it is closed. **(4.9.4.1)**

If a file renamed by the **rename** function exists before a call to the function, the file is renamed anyway. **(4.9.4.2)**

The output for **%p** conversion in the **fprintf** and **fscanf** functions is a sequence of printable characters in an integer representation. **(4.9.6.1 and 4.9.6.2)**

In the **fscanf** function, a - (hyphen) character that is neither the first nor the last character in the scan list for **%[ ]** conversion is interpreted as indicating a range of characters in the scan list. For example, you can express **[0123456789]** as **[0-9]**. The - stands for itself whenever it is the first or the last character in the scan list. **(4.9.6.2)**

### **Temporary Files (tmpfile Function)**

An open temporary file created by the **tmpfile** function is automatically deleted if the program terminates abnormally. **(4.9.4.3)**

### **errno Variable**

The **errno** variable is set to the value of the macro **EINVAL (22)** by the **fgetpos** or **ftell** function on failure. **(4.9.9.1 and 4.9.9.4)**

The  **perror** function writes a message on the standard error output that describes the last error encountered by a system call or library function. **(4.9.10.4)**

The error message includes the name of the program that caused the error followed by a : (colon), a blank, the message string, and a new-line character. The error number is taken from the global variable **errno**, which is set when an error occurs, but is not cleared when a successful call is made.

The following table lists the content of the error message strings.

| Error Number | Message Text                     | Error Number | Message Text                             |
|--------------|----------------------------------|--------------|------------------------------------------|
| 00           | Error 0 occurred.                | 52           | Missing file or file system              |
| 01           | Not owner                        | 53           | Requests blocked by Administrator        |
| 02           | No such file or directory        | 54           | Operation would block                    |
| 03           | No such process                  | 55           | Operation now in progress                |
| 04           | Interrupted system call          | 56           | Operation already in progress            |
| 05           | I/O error                        | 57           | Socket operation on non-socket           |
| 06           | No such device or address        | 58           | Destination address required             |
| 07           | Arg list too long                | 59           | Message too long                         |
| 08           | Exec format error                | 60           | Protocol wrong type for socket           |
| 09           | Bad file number                  | 61           | Protocol not available                   |
| 10           | No child processes               | 62           | Protocol not supported                   |
| 11           | Resource temporarily unavailable | 63           | Socket type not supported                |
| 12           | Not enough space                 | 64           | Operation not supported on socket        |
| 13           | Permission denied                | 65           | Protocol family not supported            |
| 14           | Bad address                      | 66           | Address not supported by protocol family |
| 15           | Block device required            | 67           | Address already in use                   |
| 16           | Device busy                      | 68           | Cannot assign requested address          |
| 17           | File exists                      | 69           | Network is down                          |
| 18           | Cross-device link                | 70           | Network is unreachable                   |
| 19           | No such device                   | 71           | Network dropped connection on reset      |
| 20           | Not a directory                  | 72           | Software caused connection abort         |
| 21           | Is a directory                   | 73           | Connection reset by peer                 |
| 22           | Invalid argument                 | 74           | No buffer space available                |
| 23           | File table overflow              | 75           | Socket is already connected              |
| 24           | Too many open files              | 76           | Socket is not connected                  |
| 25           | Not a typewriter                 | 77           | Cannot send after socket shutdown        |
| 26           | Text file busy                   | 78           | Connection timed out                     |
| 27           | File too large                   | 79           | Connection refused                       |
| 28           | No space left on device          | 80           | Host is down                             |
| 29           | Illegal seek                     | 81           | No route to host                         |
| 30           | Read-only file system            | 82           | Restart the system call                  |
| 31           | Too many links                   | 83           | Too many processes                       |
| 32           | Broken pipe                      | 84           | Too many users                           |
| 33           | Argument out of domain           | 85           | Too many levels of symbolic links        |
| 34           | Result too large                 | 86           | File name too long                       |
| 35           | No message of desired type       | 87           | Directory not empty                      |

| Error Number | Message Text                      | Error Number | Message Text                                             |
|--------------|-----------------------------------|--------------|----------------------------------------------------------|
| 36           | Identifier removed                | 88           | Disk quota exceeded                                      |
| 37           | Channel number out of range       | 93           | Item is not local to host                                |
| 38           | Level 2 not synchronized          | 109          | Function not implemented                                 |
| 39           | Level 3 halted                    | 110          | Media surface error                                      |
| 40           | Level 3 reset                     | 111          | I/O completed, but needs relocation                      |
| 41           | Link number out of range          | 112          | No attribute found                                       |
| 42           | Protocol driver not attached      | 113          | Security Authentication Denied                           |
| 43           | No CSI structure available        | 114          | Not a Trusted Program. Too many references: can't splice |
| 44           | Level 2 halted                    | 115          | Invalid wide character                                   |
| 45           | Deadlock condition if locked      | 116          | Asynchronous I/O canceled                                |
| 46           | Device not ready                  | 117          | Out of STREAMS resources                                 |
| 47           | Write-protected media             | 118          | System call timed out                                    |
| 48           | Unformatted or incompatible media | 119          | Next message has wrong type                              |
| 49           | No locks available                | 120          | Error in protocol                                        |
| 50           | Cannot Establish Connection       | 121          | No message on stream head read queue                     |
| 51           | Connection Down                   | 122          | fd not associated with a stream                          |

**Note:** Messages 89 to 92 and 94 to 108 are reserved for future use.

### Memory (calloc, malloc, and realloc Functions)

If the size requested is zero, the **calloc**, **malloc**, and **realloc** functions return a null pointer. **(4.10.3)**

### abort Function

When the **abort** function is called, open and temporary files are closed, and temporary files are erased. **(4.10.4.1)**

### exit Function

If the value of the argument to the **exit** function is other than zero, **EXIT\_SUCCESS**, or **EXIT\_FAILURE** the process ends and returns the value as the return code. **(4.10.4.3)**

### getenv Function

Environment names are defined by the underlying shell being used. **(4.10.4.4)**

The environment list obtained by a call to the **getenv** function is altered in the following manner **(4.10.4.4)**:

1. The **getenv** function searches the environment list for a string of the form *name=value*.
2. The **getenv** function returns a pointer to the value in the current environment, if such a string is present. If such a string is not present, a **NULL** pointer is returned.
3. A call to **putenv** is used. The prototype for **putenv** is:

```
int putenv(char *)
```

4. The string format is *name=value*, where *name* is the environment variable and *value* is the new value for it.

### system Function

The underlying command shell determines the format of the string that is passed to the **system**. (4.10.4.5)

The string that is passed to the **system** function goes directly to the current shell. Then the shell command interprets it as a command and runs it. (4.10.4.5)

**Note:** The **system** function runs only Bourne Shell (**bash**) commands. The results are unpredictable if the string parameter is not a valid **bash** command.

### strerror Function

The format of the error message output of the **strerror** function is the same as for **perror**. (4.11.6.2)

The contents of the error message strings returned by a call to the **strerror** function is the same as for **perror**. (4.11.6.2)

### Related Reading

"Implementation-Defined Behavior" on page 445

"Appendix I. Implementation Dependencies Overview" on page 445

## Implementation Dependency - Locale-Specific Behavior (F.4)

The C for AIX compiler defines the following locale-specific behavior:

- The local time zone and daylight saving time are set by the operating system. (4.12.1)
- The era for the **clock** function starts when the process containing the program is started. (4.12.2.1)
- The locale determines the characters of the execution set. Some locales contain characters other than those required by the C standard. (2.2.1)
- The direction of printing is left to right. (2.2.2)
- The decimal point character is locale-specific. For the United States English locale, it is the period. (4.1.1)
- The format of the time and date is specified by the format parameter of the **strftime** function. It is a character string containing two types of objects: plain characters that are simply placed in the output string, and conversion specifications for presenting the time and date in a readable form. Each conversion specification is a sequence of this form (4.12.3.5),

>>-%  
┌──────────┬──────────┬──────────┐ type ──><  
└──hyphen──┘ width ─┘ .precision ─┘

where:

|                  |                                                                                                           |
|------------------|-----------------------------------------------------------------------------------------------------------|
| <i>hyphen</i>    | Is the - character. It puts padding on the right of the converted value instead of on the left if needed. |
| <i>width</i>     | Is the minimum field width.                                                                               |
| <i>precision</i> | Is the maximum field width.                                                                               |

*type*

Is specified by one or two conversion characters. The characters and their meanings are:

|           |                                                                                               |
|-----------|-----------------------------------------------------------------------------------------------|
| <b>%D</b> | Represents the date in the format <b>%m/%d/%y</b> (for example, 01/31/92)                     |
| <b>%e</b> | Represents the day of the month as a decimal number (1 to 31), with a leading space if needed |
| <b>%E</b> | Represents the combined alternative era year and name for the locale, in <b>%o %N</b> format  |
| <b>%h</b> | Represents the abbreviated month name for the locale (for example, Jan)                       |
| <b>%n</b> | Is a new-line character                                                                       |
| <b>%N</b> | Represents the alternative era name for the locale                                            |
| <b>%o</b> | Represents the alternative era year for the locale                                            |
| <b>%r</b> | 12 hour clock format with AM/PM notation, which is the same as \$I:\$M:\$S [AM PM]            |
| <b>%t</b> | Is a tab character                                                                            |
| <b>%T</b> | 24 hour clock time in the format <b>%H:%M:%S</b> (for example, 16:55:15)                      |

Refer to the *AIX Version 4 Technical Reference, Volumes 1 and 2: Base Operating System and Extensions* for more information about time and date formatting and the **strftime** function.

### Character Testing and Case Mapping

The execution character set follows the ASCII collating sequence (**4.3 and 4.11.4.4**):

- The control characters representing horizontal tab, vertical tab, and form feed (ASCII 0-31)
- The space character
- ! " # \$ % & ' ( ) \* + , - . /
- 0 1 2 3 4 5 6 7 8 9
- : ; <=> ? @
- A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
- [ \ ] ^ \_ `
- a b c d e f g h i j k l m n o p q r s t u v w x y z
- { | } ~
- delete

### Related Reading

"Implementation-Defined Behavior" on page 445

"Appendix I. Implementation Dependencies Overview" on page 445



## Type Conversions

Type conversions are implementation dependent. Tables on this page summarize type conversions of arithmetic types. Arithmetic types include **signed** and **unsigned** integral types (**char**, **int**, **short** and **long**) in addition to **float**, **double**, and **long double** types.

- Type Conversions - Signed Integer Types (table) (page 460)
- Type Conversions - Unsigned Integer Types (table) (page 462)
- Type Conversions - Floating Point Types (table) (page 463)
- Converting Pascal String Literals (page 464)
- Integral Promotion (page 464)

| Type Conversions—Signed Integer Types |                                                                      |                                                                       |                                                                                  |                                                                                   |                                                                                        |
|---------------------------------------|----------------------------------------------------------------------|-----------------------------------------------------------------------|----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| To:                                   | signed char                                                          | signed short                                                          | signed int                                                                       | signed long                                                                       | signed long long                                                                       |
| From:                                 |                                                                      |                                                                       |                                                                                  |                                                                                   |                                                                                        |
| <b>signed char</b>                    | None                                                                 | Sign extend                                                           | Sign extend                                                                      | Sign extend                                                                       | Sign extend                                                                            |
| <b>signed short</b>                   | Preserve low-order bytes                                             | None                                                                  | sign extend                                                                      | sign extend                                                                       | sign extend                                                                            |
| <b>signed int</b>                     | Preserve low-order bytes                                             | Preserve low-order bytes                                              | None                                                                             | Preserve bit pattern                                                              | sign extend                                                                            |
| <b>signed long</b>                    | Preserve low-order bytes                                             | Preserve low-order bytes                                              | Preserve low-order bytes                                                         | None                                                                              | sign extend                                                                            |
| <b>signed long long</b>               | Preserve low-order bytes                                             | Preserve low-order bytes                                              | Preserve low-order bytes                                                         | Preserve low-order bytes                                                          | None                                                                                   |
| <b>unsigned char</b>                  | Preserve bit pattern: high-order bit becomes sign bit                | Zero extend                                                           | Zero extend                                                                      | Zero extend                                                                       | Zero extend                                                                            |
| <b>unsigned short</b>                 | Preserve low-order bytes                                             | Preserve bit pattern: high-order bit becomes sign bit                 | Zero extend                                                                      | Zero extend                                                                       | Zero extend                                                                            |
| <b>unsigned int</b>                   | Preserve low-order bytes                                             | Preserve low-order bytes                                              | Preserve bit pattern: high-order bit becomes sign bit                            | Preserve bit pattern: high-order bit becomes sign bit                             | Zero extend                                                                            |
| <b>unsigned long</b>                  | Preserve low-order bytes                                             | Preserve low-order bytes                                              | Preserve bit pattern: high-order bit becomes sign bit                            | Preserve bit pattern: high-order bit becomes sign bit                             | Zero extend                                                                            |
| <b>unsigned long long</b>             | Preserve low-order bytes                                             | Preserve low-order bytes                                              | Preserve low-order bytes                                                         | Preserve low-order bytes                                                          | Preserve bit pattern: high-order bit becomes sign bit                                  |
| <b>float</b>                          | Convert to <b>int</b> , and convert <b>int</b> to <b>signed char</b> | Convert to <b>int</b> , and convert <b>int</b> to <b>signed short</b> | Truncate at decimal. if result is too large for <b>int</b> , result is undefined | Truncate at decimal. if result is too large for <b>long</b> , result is undefined | Truncate at decimal. if result is too large for <b>long long</b> , result is undefined |

|                    |                                                                      |                                                                       |                                                                                  |                                                                                   |                                                                                        |
|--------------------|----------------------------------------------------------------------|-----------------------------------------------------------------------|----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| <b>double</b>      | Convert to <b>int</b> , and convert <b>int</b> to <b>signed char</b> | Convert to <b>int</b> , and convert <b>int</b> to <b>signed short</b> | Truncate at decimal. if result is too large for <b>int</b> , result is undefined | Truncate at decimal. if result is too large for <b>long</b> , result is undefined | Truncate at decimal. if result is too large for <b>long long</b> , result is undefined |
| <b>long double</b> | Convert to <b>int</b> , and convert <b>int</b> to <b>signed char</b> | Convert to <b>int</b> , and convert <b>int</b> to <b>signed short</b> | Truncate at decimal. if result is too large for <b>int</b> , result is undefined | Truncate at decimal. if result is too large for <b>long</b> , result is undefined | Truncate at decimal. if result is too large for <b>long long</b> , result is undefined |

| Type Conversions—Unsigned Integer Types |                                                                        |                                                                                           |                                                                           |                                                                              |                                                                                             |
|-----------------------------------------|------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|---------------------------------------------------------------------------|------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|
| To:                                     | unsigned char                                                          | unsigned short                                                                            | unsigned int                                                              | unsigned long                                                                | unsigned long long                                                                          |
| From:                                   |                                                                        |                                                                                           |                                                                           |                                                                              |                                                                                             |
| <b>signed char</b>                      | Preserve bit pattern: sign function of sign bit lost                   | Sign extend to <b>short</b> , and convert to <b>unsigned short</b>                        | Sign extend to <b>int</b> , and convert <b>int</b> to <b>unsigned int</b> | Sign extend to <b>long</b> , and convert <b>long</b> to <b>unsigned long</b> | Sign extend to <b>long long</b> , and convert <b>long long</b> to <b>unsigned long long</b> |
| <b>short</b>                            | Preserve low-order byte                                                | Preserve bit pattern: sign function of sign bit lost                                      | Sign extend to <b>int</b> , and convert <b>int</b> to <b>unsigned int</b> | Sign extend to <b>long</b> , and convert <b>long</b> to <b>unsigned long</b> | Sign extend to <b>long long</b> , and convert <b>long long</b> to <b>unsigned long long</b> |
| <b>int</b>                              | Preserve low-order byte                                                | Preserve low-order byte                                                                   | Preserve bit pattern: sign function of sign bit lost                      | Preserve bit pattern: sign function of sign bit lost                         | Sign extend to <b>long long</b> , and convert <b>long long</b> to <b>unsigned long long</b> |
| <b>long</b>                             | Preserve low-order byte                                                | Preserve low-order byte                                                                   | Preserve bit pattern: sign function of sign bit lost                      | Preserve bit pattern: sign function of sign bit lost                         | Sign extend to <b>long long</b> , and convert <b>long long</b> to <b>unsigned long long</b> |
| <b>long long</b>                        | Preserve low-order byte                                                | Preserve low-order byte                                                                   | Preserve low-order byte                                                   | Preserve low-order byte                                                      | Preserve bit pattern: sign function of sign bit lost                                        |
| <b>unsigned char</b>                    | None                                                                   | Zero extend                                                                               | Zero extend                                                               | Zero extend                                                                  | Zero extend                                                                                 |
| <b>unsigned short</b>                   | Preserve low-order byte                                                | None                                                                                      | Zero extend                                                               | Zero extend                                                                  | Zero extend                                                                                 |
| <b>unsigned int</b>                     | Preserve low-order bytes                                               | Preserve low-order bytes                                                                  | None                                                                      | Preserve bit pattern                                                         | Zero extend                                                                                 |
| <b>unsigned long</b>                    | Preserve low-order bytes                                               | Preserve low-order bytes                                                                  | Preserve bit pattern                                                      | None                                                                         | Zero extend                                                                                 |
| <b>unsigned long long</b>               | Preserve low-order bytes                                               | Preserve low-order bytes                                                                  | Preserve low-order bytes                                                  | Preserve low-order bytes                                                     | None                                                                                        |
| <b>float</b>                            | Convert to <b>int</b> , and convert <b>int</b> to <b>unsigned char</b> | Convert to <b>unsigned int</b> , and convert <b>unsigned int</b> to <b>unsigned short</b> | Truncate; if result is negative or too large, the result is undefined     | Truncate; if result is negative or too large, the result is undefined        | Truncate; if result is negative or too large, the result is undefined                       |

|                    |                                                                        |                                                                                           |                                                                       |                                                                       |                                                                       |
|--------------------|------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|-----------------------------------------------------------------------|-----------------------------------------------------------------------|-----------------------------------------------------------------------|
| <b>double</b>      | Convert to <b>int</b> , and convert <b>int</b> to <b>unsigned char</b> | Convert to <b>unsigned int</b> , and convert <b>unsigned int</b> to <b>unsigned short</b> | Truncate; if result is negative or too large, the result is undefined | Truncate; if result is negative or too large, the result is undefined | Truncate; if result is negative or too large, the result is undefined |
| <b>long double</b> | Convert to <b>int</b> , and convert <b>int</b> to <b>unsigned char</b> | Convert to <b>unsigned int</b> , and convert <b>unsigned int</b> to <b>unsigned short</b> | Truncate; if result is negative or too large, the result is undefined | Truncate; if result is negative or too large, the result is undefined | Truncate; if result is negative or too large, the result is undefined |

| <b>Type Conversions—Floating-point Types</b> |                                                                                                                     |                                                                                                                      |                                                                                                                           |
|----------------------------------------------|---------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| <b>To:</b>                                   | <b>float</b>                                                                                                        | <b>double</b>                                                                                                        | <b>long double</b>                                                                                                        |
| <b>From:</b>                                 |                                                                                                                     |                                                                                                                      |                                                                                                                           |
| <b>signed char</b>                           | Sign extend to <b>int</b> , and convert <b>int</b> to <b>float</b>                                                  | Sign extend to <b>int</b> , and convert <b>int</b> to <b>double</b>                                                  | Sign extend to <b>int</b> , and convert <b>int</b> to <b>long double</b>                                                  |
| <b>signed short</b>                          | Sign extend to <b>int</b> , and convert <b>int</b> to <b>float</b>                                                  | Sign extend to <b>int</b> , and convert <b>int</b> to <b>double</b>                                                  | Sign extend to <b>int</b> , and convert <b>int</b> to <b>long double</b>                                                  |
| <b>signed int</b>                            | Represent as <b>float</b> ; if the <b>int</b> cannot be represented exactly, some loss of precision may occur       | Represent as <b>double</b> ; if the <b>int</b> cannot be represented exactly, some loss of precision may occur       | Represent as <b>long double</b> ; if the <b>int</b> cannot be represented exactly, some loss of precision may occur       |
| <b>signed long</b>                           | Represent as <b>float</b> ; if the <b>long</b> cannot be represented exactly, some loss of precision may occur      | Represent as <b>double</b> ; if the <b>long</b> cannot be represented exactly, some loss of precision may occur      | Represent as <b>long double</b> ; if the <b>long</b> cannot be represented exactly, some loss of precision may occur      |
| <b>signed long long</b>                      | Represent as <b>float</b> ; if the <b>long long</b> cannot be represented exactly, some loss of precision may occur | Represent as <b>double</b> ; if the <b>long long</b> cannot be represented exactly, some loss of precision may occur | Represent as <b>long double</b> ; if the <b>long long</b> cannot be represented exactly, some loss of precision may occur |
| <b>unsigned char</b>                         | Sign extend to <b>int</b> , and convert <b>int</b> to <b>float</b>                                                  | Sign extend to <b>int</b> , and convert <b>int</b> to <b>double</b>                                                  | Sign extend to <b>int</b> , and convert <b>int</b> to <b>long double</b>                                                  |
| <b>unsigned short</b>                        | Sign extend to <b>int</b> , and convert <b>int</b> to <b>float</b>                                                  | Sign extend to <b>int</b> , and convert <b>int</b> to <b>double</b>                                                  | Sign extend to <b>int</b> , and convert <b>int</b> to <b>long double</b>                                                  |
| <b>unsigned int</b>                          | Represent as <b>float</b> ; if the <b>int</b> cannot be represented exactly, some loss of precision may occur       | Represent as <b>double</b> ; if the <b>int</b> cannot be represented exactly, some loss of precision may occur       | Represent as <b>long double</b> ; if the <b>int</b> cannot be represented exactly, some loss of precision may occur       |
| <b>unsigned long</b>                         | Represent as <b>float</b> ; if the <b>long</b> cannot be represented exactly, some loss of precision may occur      | Represent as <b>double</b> ; if the <b>long</b> cannot be represented exactly, some loss of precision may occur      | Represent as <b>long double</b> ; if the <b>long</b> cannot be represented exactly, some loss of precision may occur      |
| <b>unsigned long long</b>                    | Represent as <b>float</b> ; if the <b>long long</b> cannot be represented exactly, some loss of precision may occur | Represent as <b>double</b> ; if the <b>long long</b> cannot be represented exactly, some loss of precision may occur | Represent as <b>long double</b> ; if the <b>long long</b> cannot be represented exactly, some loss of precision may occur |
| <b>float</b>                                 | None                                                                                                                | Convert to <b>double</b>                                                                                             | Convert to <b>long double</b>                                                                                             |
| <b>double</b>                                | Represent as <b>float</b> ; if result is too large, result is undefined                                             | None                                                                                                                 | Convert to <b>long double</b>                                                                                             |

|                    |                         |                                                                                                              |      |
|--------------------|-------------------------|--------------------------------------------------------------------------------------------------------------|------|
| <b>long double</b> | Convert to <b>float</b> | Represent as <b>double</b> ; if result is too large to be represented as <b>double</b> , result is undefined | None |
|--------------------|-------------------------|--------------------------------------------------------------------------------------------------------------|------|

**Converting Pascal String Literals**The **-qmacpstr** option converts Pascal string literals of the form “\pABC” into null-terminated strings where the first byte contains the length of the string.

**Integral Promotion**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**.

The **-qupconv** option promotes any **unsigned** type smaller than an **int** to an **unsigned int** instead of to an **int**.

#### Related Reading

“Appendix I. Implementation Dependencies Overview” on page 445  
“macpstr” on page 295  
“-upconv” on page 334

---

## Synchronization of Stores and Loads to I/O Space

Stores and loads to I/O space must be separated from other store and load operations by synchronization instructions. These synchronization instructions allow the compiler to synchronize the execution of the code block.

The C for AIX compiler provides two built-in functions that you can use to specify the type of synchronization, and where and how it should take place.

For C source programs, the two functions are:

- **extern void \_\_iospace\_eieio(void);**
- **extern void \_\_iospace\_sync(void);**

Use the function **\_\_iospace\_eieio** (Enforced In-order Execution for I/O) when you want to synchronize cache-inhibited memory-mapped I/O.

Use the function **\_\_iospace\_sync** (I/O Space Synchronization) when all the multiple dispatch capabilities of the processor must be synchronized.

To use these built-in functions effectively, you should use pointers to **volatile** to access device registers, and then place either of these built-in functions at the completion of this access section of your code.

#### Related Reading

“Example of Reading and Writing to Mapped-to-I/O Space” on page 465  
“Example of Multiple Writes to a Single Register” on page 465  
“Appendix I. Implementation Dependencies Overview” on page 445

## Example of Multiple Writes to a Single Register

```
/*
Consider a machine with a single register.
This register must be fed a sequence of four
commands in exact order to form a complete instruction.
To prevent reordering of the command sequence,
the __iospace_eieio(); function is used.
The sequence of commands is:
    1. load address
    2. low-order byte data
    3. high-order byte data
    4. load word command
Each set of instruction data needs to be synchronized.
*/
typedef unsigned long data;
#define COMMAND_REG ((volatile data *const) 0xFFFFF20)
#define LOAD_ADDRESS 0xFFFF2000
#define LOAD_HIGH_DATA 0xFF
#define LOAD_LOW_DATA 0xA0
#define LOAD_COMMAND 0x10
void send_load_command()
{
    *COMMAND_REG=LOAD_ADDRESS;
    __iospace_eieio(); /* ensure the stores are ordered */
    *COMMAND_REG=LOAD_LOW_DATA;
    __iospace_eieio(); /* ensure the stores are ordered */
    *COMMAND_REG=LOAD_HIGH_DATA;
    __iospace_eieio(); /* ensure the stores are ordered */
    *COMMAND_REG=LOAD_COMMAND;
    __iospace_eieio(); /* ensure the stores are ordered */
}

```

### Related Reading

“Example of Reading and Writing to Mapped-to-I/O Space”  
“Appendix I. Implementation Dependencies Overview” on page 445  
“Synchronization of Stores and Loads to I/O Space” on page 464

## Example of Reading and Writing to Mapped-to-I/O Space

```
/* The following example assumes a special machine
with three hardware locations.
There is an input data buffer at 0xFFFFF20
There is an output data buffer at 0xFFFFF24
There is a control data buffer at 0xFFFFF28
The third-least-significant bit set in the control
buffer indicates there is data in the input buffer.
This bit is unset when the data is read out of the
input buffer by the program.
The second-least-significant bit set in the control
buffer indicates the output device is ready to accept
output data.
This bit is unset when the data is placed in the
output buffer and is read out.
*/
typedef unsigned long datatype, controltype, counttype;
#define CONTROLLER ((const volatile controltype *const) 0xFFFFF28)
#define INPUT_BUF ((const volatile datatype *const) 0xFFFFF20)
#define OUTPUT_BUF ((volatile datatype *const) 0xFFFFF24)
#define INPUT_READY_BIT 0x4
#define OUTPUT_READY_BIT 0x2
#define INPUT_READY (~(*CONTROLLER) & INPUT_READY_BIT)
#define OUTPUT_READY ((*CONTROLLER) & OUTPUT_READY_BIT)
/*
The function copy_data copies data from the input buffer
to the output buffer until an input value of 0 is seen.
*/

```

The number of characters copied is returned.

```
*/
/* Make count of the input global so that other CPUs can access it */
counttype volatile count=0;
counttype copy_data()
{
    datatype temp;
    for(;;) {
        while (!INPUT_READY);          /* Wait for input    */
        temp=*INPUT_BUF;
        __iospace_eieio(); /*      synchronize the I/O */
        if (temp==0) return count;
        while (!OUTPUT_READY);         /* Wait for output   */
        *OUTPUT_BUF=temp;
        __iospace_eieio(); /*      synchronize the I/O */
        count++;
        __iospace_sync(); /*      synchronize the CPU */
        /*      for count to allow */
        /*      other CPUs to */
        /*      access it */
    }
}
```

### Related Reading

“Example of Multiple Writes to a Single Register” on page 465

“Appendix I. Implementation Dependencies Overview” on page 445

“Synchronization of Stores and Loads to I/O Space” on page 464

---

## Appendix J. C for AIX and XL C Compatibility

This appendix discusses compatibility issues between the C for AIX compiler and the XL C compiler.

The C for AIX compiler is not fully compatible with XL C: it is a stricter compiler than XL C. The following are the differences:

1. C for AIX implements tighter ANSI conformance. For example, the following code segments are accepted by XL C, but not by C for AIX:

*Example 1: Mixing K&R-style and ANSI-style function prototypes.*

```
void func();  
void func(float f){ ..... }
```

Accepting this code leads to runtime problems since the float parameter in the definition is first promoted to double (default argument promotions). When the float argument is passed in, the wrong size registers are taken off the stack.

*Example 2: Null dimension multi-dimensional arrays.*

Arrays of incomplete arrays are not allowed, not even on parameters:

```
void f(int p[] []);
```

All dimensions except the first must be specified for a multi-dimensional array. In the above example, p is defined to be an incomplete array of an incomplete type (an incomplete array of an incomplete array of int).

*Example 3: Tags introduced at parameter scope are not exported to the enclosing non-parameter scope.*

```
int f(struct a *);  
struct a { int a; };  
int f(struct a* i ) { return i->a; }
```

The type struct a was introduced in a parameter list, and will go out of scope at the end of the function declaration or definition.

2. The C for AIX compiler may differ from XL C in implementation-defined parts of the language.
  - Apparent order of evaluation of arguments to functions with assignments will generate different results with XL C 1.3 and with C for AIX. Other industry compilers may generate results with even greater differences. This code is not portable and should be avoided.

```
#include <stdio.h>  
void func(int i, int j, int k)  
{  
    printf("i = %d, j = %d, k = %d\n",i,j,k);  
}  
main()  
{  
    int r=1;  
    int c=0;  
    func(c=r,r,r=4);  
}
```

With XL C 1.3 the results are:

```
i = 1 j = 1 k = 4
```

With C for AIX the results are:

```
i = 1 j = 4 k = 4
```

- Preprocessor differences include:

- Output preserves coordinates of each token.
  - No redundant **#line** directives or multiple blank lines in output.
  - Erroneous/incomplete macro invocations are expanded anyway. This includes:
    - Extra arguments (ignored)
    - Empty arguments
    - Missing arguments (treated as empty)
    - Incomplete argument list (treated as completed)
3. C for AIX differs from XL C where provisions for future extensions are implemented.
- C for AIX requires that **#pragma options align=***suboption* appears before the structure definition. XL C allows the **#pragma** to appear anywhere before the last closing brace of the structure definition.
  - **long long** type is supported, even under ANSI mode. **long long** is recognized by all of the invocation commands except for **ic89** and **c89**, which disable it by specifying the **-qolonglong** option.
  - Invoking the compiler does not define **\_ANSI\_C\_SOURCE** (unless you use the **ic89** or **c89** invocation commands, or unless you include **standards.h**). This is because there are many useful definitions in header files that are not detected when **\_ANSI\_C\_SOURCE** is in effect. You can define **\_ANSI\_C\_SOURCE** by supplying **-D\_ANSI\_C\_SOURCE** at compiler invocation.



---

## Appendix K. National Languages Support in the C for AIX Compiler

This and related pages summarize the national language support (NLS) specific to the C for AIX Compiler.

**Note:** You must specify the **-qmbcs** option to use multibyte characters anywhere in your program.

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. See Restrictions (page 471) for more information.

### Related Reading

“Converting Files Containing Multibyte Data to New Code Pages”

“Where Multibyte Characters Are Supported”

“mbcs, ducs” on page 301

---

## Converting Files Containing Multibyte Data to New Code Pages

If you have installed new code pages on your system, you must 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 Version 4 Commands Reference*. Using the NLS code set converters with the **iconv** command is described in “Converters Overview for Programming” in the *AIX Version 4 General Programming Concepts*.

### Related Reading

“Appendix K. National Languages Support in the C for AIX Compiler”

“Where Multibyte Characters Are Supported”

---

## Where Multibyte Characters Are Supported

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'
```

String literals and character constants are described in *AIX Version 4: Language Reference*. The # and ## operators, preprocessor directives, and macro definitions are also described there.

### Preprocessor Directives

The following preprocessor directives permit multibyte-character constants and string literals:

- **#define**
- **#pragma comment**
- **#include**

These and other preprocessor directives are described in *AIX Version 4: Language Reference*.

### 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 **-D** *name=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 **-qdbc** 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 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. Comments are described in *AIX Version 4: Language Reference*.

### 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 Reading

“Appendix K. National Languages Support in the C for AIX Compiler” on page 469

“Where Multibyte Characters Are Supported” on page 469

“B” on page 239

“D” on page 250

“F” on page 259

“I” on page 272

“L” on page 285

“I” on page 286

“O” on page 305

“list” on page 291

“mbcs, dbcs” on page 301

“ro” on page 317

“source” on page 322



## Appendix L. C for AIX Files

You can install the C for AIX. as an optional software product of the AIX for RISC System/6000. Installing the compiler places the following files on your system:

|                                                                                                                                                                        |                                                                                                                                                                                                                                                               |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| C for AIX README file, which contains important information not included in other documentation. <i>Read this file before you use the compiler for the first time.</i> | /usr/vac/xlC/README.C                                                                                                                                                                                                                                         |
| C front end                                                                                                                                                            | /usr/vac/exe/xlcentry                                                                                                                                                                                                                                         |
| Help file                                                                                                                                                              | /usr/vac/exe/default_msg/vac.help                                                                                                                                                                                                                             |
| C preprocessor                                                                                                                                                         | /usr/vac/exe/xlCcpp                                                                                                                                                                                                                                           |
| Disassembler                                                                                                                                                           | /usr/vac/exe/dis                                                                                                                                                                                                                                              |
| Interprocedural Analysis tool                                                                                                                                          | /usr/vac/exe/ipa                                                                                                                                                                                                                                              |
| Code generator                                                                                                                                                         | /usr/vac/exe/xlCcode<br>/usr/vac/exe/bolt                                                                                                                                                                                                                     |
| Default message catalogs                                                                                                                                               | /usr/vac/exe/default_msg/vacdmsg.cat<br>/usr/vac/exe/default_msg/vacfe.cat<br>/usr/vac/exe/default_msg/vacimsg.cat<br>/usr/vac/exe/default_msg/vacsmsg.cat                                                                                                    |
| C driver programs                                                                                                                                                      | /usr/vac/bin/xlc<br>/usr/vac/bin/xlc128<br>/usr/vac/bin/xlc_r<br>/usr/vac/bin/cc<br>/usr/vac/bin/cc128<br>/usr/vac/bin/cc_r<br>/usr/vac/bin/cleanpdf<br>/usr/vac/bin/replaceCSET<br>/usr/vac/bin/resetpdf<br>/usr/vac/bin/restoreCSET<br>/usr/vac/bin/showpdf |
| Precompiled header support                                                                                                                                             | /usr/vac/lib/compmalloc.o                                                                                                                                                                                                                                     |
| Memory debug support                                                                                                                                                   | /usr/vac/lib/libhm.a<br>/usr/vac/lib/libhm_r.a<br>/usr/vac/lib/libhmd.a<br>/usr/vac/lib/libhmd_r.a<br>/usr/vac/lib/libhu.a<br>/usr/vac/lib/libhu_r.a<br>/usr/vac/include/stdlib.h<br>/usr/vac/include/string.h<br>/usr/vac/include/umalloc.h                  |
| Profile-directed feedback library                                                                                                                                      | /usr/vac/lib/libpdf.a                                                                                                                                                                                                                                         |
| Profiling library                                                                                                                                                      | /usr/vac/lib/profiled                                                                                                                                                                                                                                         |
| Configuration file                                                                                                                                                     | /etc/vac.cfg                                                                                                                                                                                                                                                  |
| Links to /usr/vac/bin                                                                                                                                                  | /usr/bin/xlc<br>/usr/bin/xlc128<br>/usr/bin/xlc_r<br>/usr/bin/c89<br>/usr/bin/cc<br>/usr/bin/cc128<br>/usr/bin/cc_r                                                                                                                                           |

## C for AIX Compiler Invocation Commands

All invocation command files are placed in `/usr/vac/bin`. You may add this directory to the beginning of your `PATH` environment variable to make these commands available to you.

The default links in `/usr/bin` (`xlc`, `cc`, `c89`, etc) that point to `/usr/vac/bin` of C for AIX are optional. They are created at the discretion of the product installer using `replaceCSET`.

---

## etc/vac.cfg - Default Configuration File

The C for AIX Compiler includes two versions of the default configuration file. When you install the C for AIX Compiler, only one of these configuration files is linked to `/etc/vac.cfg` according to the following criteria:

- If the level of your operating system is AIX 4.3 or higher, `/etc/vac.cfg` is linked to the “vac.cfg.43 Compiler Configuration File” on page 477 default configuration file
- If the level of your operating system is earlier than AIX 4.3, `/etc/vac.cfg` is linked to the “vac.cfg.41 Compiler Configuration File” default configuration file.

### Related Reading

“Compiler Options” on page 10

“Invoking the Compiler” on page 8

“Specifying Compiler Options on the Command Line” on page 10

“Specifying Compiler Options in Your Program Source Files” on page 12

“Specifying Compiler Options in a Configuration File” on page 13

“Resolving Conflicting Compiler Options” on page 217

“Compiler Options and Their Defaults” on page 218

## vac.cfg.41 Compiler Configuration File

```
*
* COMPONENT_NAME: (CC) C for AIX Compiler
*
* FUNCTIONS: C/C++ Configuration file
*
* ORIGINS: 27
*
* (C) COPYRIGHT International Business Machines Corp. 1991, 1999
* All Rights Reserved
* Licensed Materials - Property of IBM
*
* US Government Users Restricted Rights - Use, duplication or
* disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
*
* ANSI C compiler, UNIX header files
xlc:  use      = DEFLT
      crt      = /lib/crt0.o
      mcrt     = /lib/mcrt0.o
      gcrt     = /lib/gcrt0.o
      libraries = -lc
      proflibs = -L/lib/profiled,-L/usr/lib/profiled
      options  = -qansialias
* C compiler, extended mode
cc:   use      = DEFLT
      crt      = /lib/crt0.o
      mcrt     = /lib/mcrt0.o
      gcrt     = /lib/gcrt0.o
      libraries = -lc
```

```

        proflibs = -L/lib/profiled,-L/usr/lib/profiled
        options  = -qlanglvl=extended,-qnor,-qnorconst
xlc128: use     = DEFLT
        crt      = /lib/crt0.o
        mcrt     = /lib/mcrt0.o
        gcrt     = /lib/gcrt0.o
        libraries = -lc128,-lc
        proflibs = -L/lib/profiled,-L/usr/lib/profiled
        options  = -qansialias,-qldb1128
* C compiler, extended mode
cc128: use     = DEFLT
        crt      = /lib/crt0.o
        mcrt     = /lib/mcrt0.o
        gcrt     = /lib/gcrt0.o
        libraries = -lc128,-lc
        proflibs = -L/lib/profiled,-L/usr/lib/profiled
        options  = -qlanglvl=extended,-qnor,-qnorconst,-qldb1128
* Strict ANSI compiler, ANSI headers
c89:  use     = DEFLT
        crt      = /lib/crt0.o
        mcrt     = /lib/mcrt0.o
        gcrt     = /lib/gcrt0.o
        libraries = -lc
        proflibs = -L/lib/profiled,-L/usr/lib/profiled
        options  = -D_ANSI_C_SOURCE,-qansialias,-qnoqlonglong,-qstrict_induction
* C++ compiler
xlc:  use     = DEFLT
        crt      = /usr/vacpp/lib/crt0.o
        mcrt     = /usr/vacpp/lib/mcrt0.o
        gcrt     = /usr/vacpp/lib/gcrt0.o
        libraries2 = -L/usr/vacpp/lib,-lc,-lm,-lc
        proflibs = -L/usr/vacpp/lib/profiled,-L/lib/profiled,-L/usr/lib/profiled
        hdlibs   = -L/usr/vac/lib,-lhC,-lhmd
        options  = -qansialias
* C++ compiler with 128 bit long double
xlc128: use     = DEFLT
        crt      = /usr/vacpp/lib/crt0.o
        mcrt     = /usr/vacpp/lib/mcrt0.o
        gcrt     = /usr/vacpp/lib/gcrt0.o
        libraries2 = -L/usr/vacpp/lib,-lc128,-lc,-lm,-lc128,-lc
        proflibs = -L/usr/vacpp/lib/profiled,-L/lib/profiled,-L/usr/lib/profiled
        hdlibs   = -L/usr/vac/lib,-lhC,-lhmd
        options  = -qansialias,-qldb1128
* standard c compiler aliased as xlc_r (AIX Threads)
xlc_r: use     = DEFLT
        crt      = /lib/crt0_r.o
        mcrt     = /lib/mcrt0_r.o
        gcrt     = /lib/gcrt0_r.o
        libraries = -L/usr/lib/threads,-lpthreads,-lc_r,/usr/lib/libc.a
        proflibs = -L/lib/profiled,-L/usr/lib/profiled
        hdlibs   = -L/usr/vac/lib,-lhmd_r
        options  = -qansialias,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__
* standard c compiler aliased as cc_r (AIX Threads)
cc_r:  use     = DEFLT
        crt      = /lib/crt0_r.o
        mcrt     = /lib/mcrt0_r.o
        gcrt     = /lib/gcrt0_r.o
        libraries = -L/usr/lib/threads,-lpthreads,-lc_r,/usr/lib/libc.a
        proflibs = -L/lib/profiled,-L/usr/lib/profiled
        hdlibs   = -L/usr/vac/lib,-lhmd_r
        options  = -qlanglvl=extended,-qnor,-qnorconst,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__
xlc128_r: use   = DEFLT
        crt      = /lib/crt0_r.o
        mcrt     = /lib/mcrt0_r.o
        gcrt     = /lib/gcrt0_r.o
        libraries = -L/usr/lib/threads,-lpthreads,-lc128_r,-lc128,-lc_r,/usr/lib/libc.a
        proflibs = -L/lib/profiled,-L/usr/lib/profiled

```

```

    hdlibs      = -L/usr/vac/lib,-lhmd_r
    options     = -qansialias,-qldbl128,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__
* C compiler, extended mode
cc128_r: use   = DEFLT
    crt        = /lib/crt0_r.o
    mcrt       = /lib/mcrt0_r.o
    gcrt       = /lib/gcrt0_r.o
    libraries  = -L/usr/lib/threads,-lpthreads,-lc128_r,-lc128,-lc_r,/usr/lib/libc.a
    proflibs   = -L/lib/profiled,-L/usr/lib/profiled
    hdlibs     = -L/usr/vac/lib,-lhmd_r
    options    = -qlanglvl=extended,-qnoronoconst,-qldbl128,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__
* x1C C++ compiler aliased as x1C_r (AIX Threads)
x1C_r: use    = DEFLT
    crt        = /usr/vacpp/lib/crt0_r.o
    mcrt       = /usr/vacpp/lib/mcrt0_r.o
    gcrt       = /usr/vacpp/lib/gcrt0_r.o
    options    = -qansialias,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__
    libraries2 = -L/usr/lib/threads,-L/usr/vacpp/lib,-lc_r,-lc,-lpthreads,-lc_r,-lm,/usr/lib/libc.a
    proflibs   = -L/usr/vacpp/lib/profiled,-L/lib/profiled,-L/usr/lib/profiled
    hdlibs     = -L/usr/vac/lib,-lhC_r,-lhmd_r
* C++ compiler with 128 bit long double (AIX Threads)
x1C128_r: use = DEFLT
    crt        = /usr/vacpp/lib/crt0_r.o
    mcrt       = /usr/vacpp/lib/mcrt0_r.o
    gcrt       = /usr/vacpp/lib/gcrt0_r.o
    libraries2 = -L/usr/lib/threads,-L/usr/vacpp/lib,-lc128_r,-lc128,-lc_r,-lc,-lpthreads,-lc128_r,-lc128,-lc_r,-lm,/usr/lib/libc.a
    proflibs   = -L/usr/vacpp/lib/profiled,-L/lib/profiled,-L/usr/lib/profiled
    hdlibs     = -L/usr/vac/lib,-lhC_r,-lhmd_r
    options    = -qansialias,-qldbl128,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__
* standard c compiler aliased as x1c_r4 (DCE)
x1c_r4: use   = DEFLT
    crt        = /lib/crt0_r.o
    mcrt       = /lib/mcrt0_r.o
    gcrt       = /lib/gcrt0_r.o
    libraries  = -L/usr/lib/threads,-ldcelibc_r,-ldcepthreads,-lpthreads,-lc_r,-lc
    proflibs   = -L/lib/profiled,-L/usr/lib/profiled
    hdlibs     = -L/usr/vac/lib,-lhmd_r
    options    = -qansialias,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__, -D_AIX32_THREADS=1,-D_AES_SOURCE,-D_AIX41,-I/
* standard c compiler aliased as cc_r4 (DCE)
cc_r4: use    = DEFLT
    crt        = /lib/crt0_r.o
    mcrt       = /lib/mcrt0_r.o
    gcrt       = /lib/gcrt0_r.o
    libraries  = -L/usr/lib/threads,-ldcelibc_r,-ldcepthreads,-lpthreads,-lc_r,-lc
    proflibs   = -L/lib/profiled,-L/usr/lib/profiled
    hdlibs     = -L/usr/vac/lib,-lhmd_r
    options    = -qlanglvl=extended,-qnoronoconst,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__, -D_AIX32_THREADS=1,-
x1c128_r4: use = DEFLT
    crt        = /lib/crt0_r.o
    mcrt       = /lib/mcrt0_r.o
    gcrt       = /lib/gcrt0_r.o
    libraries  = -L/usr/lib/threads,-ldcelibc_r,-ldcepthreads,-lpthreads,-lc128_r,-lc128,-lc_r,-lc
    proflibs   = -L/lib/profiled,-L/usr/lib/profiled
    hdlibs     = -L/usr/vac/lib,-lhmd_r
    options    = -qansialias,-qldbl128,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__
* C compiler, extended mode
cc128_r4: use = DEFLT
    crt        = /lib/crt0_r.o
    mcrt       = /lib/mcrt0_r.o
    gcrt       = /lib/gcrt0_r.o
    libraries  = -L/usr/lib/threads,-ldcelibc_r,-ldcepthreads,-lpthreads,-lc128_r,-lc128,-lc_r,-lc
    proflibs   = -L/lib/profiled,-L/usr/lib/profiled
    hdlibs     = -L/usr/vac/lib,-lhmd_r
    options    = -qlanglvl=extended,-qnoronoconst,-qldbl128,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__
* x1C C++ compiler aliased as x1C_r4 (DCE)
x1C_r4: use   = DEFLT
    crt        = /usr/vacpp/lib/crt0_r.o

```



```

mcrtd      = /usr/vacpp/lib/mcrt0_r.o
gcrt       = /usr/vacpp/lib/gcrt0_r.o
libraries2 = -L/usr/lib/threads,-L/usr/vacpp/lib,-lc_r,-lc,-ldcelibc_r,-ldcepthreads,-lpthreads,-lc_r,-lm,/usr/1
proflibs   = -L/usr/vacpp/lib/profiled,-L/lib/profiled,-L/usr/lib/profiled
hdlibs     = -L/usr/vac/lib,-lhC_r,-lhmd_r
options    = -qansialias,-qthreaded,-D__VACPP_MULTI__, -D_THREAD_SAFE,-D_AIX32_THREADS=1,-D_AIX41,-D_AES_SOURCE,-
* C++ compiler with 128 bit long double (AIX Threads)
xlC128_r4: use      = DEFLT
crt        = /usr/vacpp/lib/crt0_r.o
mcrtd      = /usr/vacpp/lib/mcrt0_r.o
gcrt       = /usr/vacpp/lib/gcrt0_r.o
libraries2 = -L/usr/lib/threads,-L/usr/vacpp/lib,-lc128_r,-lc128,-lc_r,-lc,-ldcelibc_r,-ldcepthreads,-lpthreads,-
proflibs   = -L/usr/vacpp/lib/profiled,-L/lib/profiled,-L/usr/lib/profiled
hdlibs     = -L/usr/vac/lib,-lhC,-lhmd
options    = -qansialias,-qldb128,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__, -D_AIX32_THREADS=1,-D_AIX41,-D_AES
* common definitions
DEFLT: cppcomp = /usr/vacpp/exe/xlCentry
ccomp    = /usr/vac/exe/xlcentry
code     = /usr/vac/exe/xlCcode
cpp      = /usr/vac/exe/xlCcpp
munch   = /usr/vacpp/exe/munch
ipa     = /usr/vac/exe/ipa
dis     = /usr/vac/exe/dis
cppfilt = /usr/vacpp/bin/c++filt
bolt    = /usr/vac/exe/bolt
as      = /bin/as
ld      = /bin/ld
xlC     = /usr/vacpp/bin/xlC
cppinc  = /usr/vacpp/include
options = -D_AIX,-D_AIX32,-D_AIX41,-D_IBMR2,-D_POWER,-bpT:0x10000000,-bpD:0x20000000
ldopt   = "b:o:e:u:R:H:Y:Z:L:T:A:k:j:"
hdlibs  = -L/usr/vac/lib,-lhmd
xlCcopt = -qansialias
smplibraries = -lxlsmp
optlibraries = -lxlopt
resexp   = /usr/vacpp/lib/res.exp
genexports = /usr/vac/bin/CreateExportList

```

## vac.cfg.43 Compiler Configuration File

```

*
* COMPONENT_NAME: (CC) C for AIX Compiler
*
* FUNCTIONS: C/C++ Configuration file
*
* ORIGINS: 27
*
* (C) COPYRIGHT International Business Machines Corp. 1991, 1999
* All Rights Reserved
* Licensed Materials - Property of IBM
*
* US Government Users Restricted Rights - Use, duplication or
* disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
*
* ANSI C compiler, UNIX header files
xlC:  use      = DEFLT
      crt      = /lib/crt0.o
      mcrtd    = /lib/mcrt0.o
      gcrt     = /lib/gcrt0.o
      libraries = -lc
      proflibs = -L/lib/profiled,-L/usr/lib/profiled
      options  = -qansialias
* C compiler, extended mode
cc:   use      = DEFLT
      crt      = /lib/crt0.o

```

```

    mcrt      = /lib/mcrt0.o
    gcrt      = /lib/gcrt0.o
    libraries = -lc
    proflibs  = -L/lib/profiled,-L/usr/lib/profiled
    options   = -qlanglvl=extended,-qnor0,-qnorconst
xlc128: use  = DEFLT
    crt      = /lib/crt0.o
    mcrt     = /lib/mcrt0.o
    gcrt     = /lib/gcrt0.o
    libraries = -lc128,-lc
    proflibs = -L/lib/profiled,-L/usr/lib/profiled
    options  = -qansialias,-qldbl128
* C compiler, extended mode
cc128: use  = DEFLT
    crt      = /lib/crt0.o
    mcrt     = /lib/mcrt0.o
    gcrt     = /lib/gcrt0.o
    libraries = -lc128,-lc
    proflibs = -L/lib/profiled,-L/usr/lib/profiled
    options  = -qlanglvl=extended,-qnor0,-qnorconst,-qldbl128
* Strict ANSI compiler, ANSI headers
c89: use  = DEFLT
    crt      = /lib/crt0.o
    mcrt     = /lib/mcrt0.o
    gcrt     = /lib/gcrt0.o
    libraries = -lc
    proflibs = -L/lib/profiled,-L/usr/lib/profiled
    options  = -D_ANSI_C_SOURCE,-qansialias,-qnoqlonglong,-qstrict_induction
* C++ compiler
xlc: use  = DEFLT
    crt      = /lib/crt0.o
    mcrt     = /lib/mcrt0.o
    gcrt     = /lib/gcrt0.o
    libraries2 = -L/usr/vacpp/lib,-lc,-lm,-lpthreads,-lc
    proflibs  = -L/usr/vacpp/lib/profiled,-L/lib/profiled,-L/usr/lib/profiled
    hdlibs    = -L/usr/vac/lib,-lhC,-lhmd
    options   = -qansialias
* C++ compiler with 128 bit long double
xlc128: use  = DEFLT
    crt      = /lib/crt0.o
    mcrt     = /lib/mcrt0.o
    gcrt     = /lib/gcrt0.o
    libraries2 = -L/usr/vacpp/lib,-lc128,-lc,-lm,-lpthreads,-lc128,-lc
    proflibs  = -L/usr/vacpp/lib/profiled,-L/lib/profiled,-L/usr/lib/profiled
    hdlibs    = -L/usr/vac/lib,-lhC,-lhmd
    options   = -qansialias,-qldbl128
* standard c compiler aliased as xlc_r (AIX Threads)
xlc_r: use  = DEFLT
    crt      = /lib/crt0.o
    mcrt     = /lib/mcrt0.o
    gcrt     = /lib/gcrt0.o
    libraries = -L/usr/lib/threads,-lpthreads,-lc
    proflibs  = -L/lib/profiled,-L/usr/lib/profiled
    hdlibs    = -L/usr/vac/lib,-lhmd
    options   = -qansialias,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__
* xlc128_r
xlc128_r: use  = DEFLT
    crt      = /lib/crt0.o
    mcrt     = /lib/mcrt0.o
    gcrt     = /lib/gcrt0.o
    libraries = -L/usr/lib/threads,-lpthreads,-lc128,-lc
    proflibs  = -L/lib/profiled,-L/usr/lib/profiled
    hdlibs    = -L/usr/vac/lib,-lhmd
    options   = -qansialias,-qldbl128,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__
* standard c compiler aliased as cc_r (AIX Threads)
cc_r: use  = DEFLT
    crt      = /lib/crt0.o

```

```

mcrtd      = /lib/mcrt0.o
gcrt       = /lib/gcrt0.o
libraries  = -L/usr/lib/threads,-lpthreads,-lc
proflibs   = -L/lib/profiled,-L/usr/lib/profiled
hdlibs     = -L/usr/vac/lib,-lhmd
options    = -qlanglvl=extended,-qnor,-qnorconst,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__
* cc128_r
cc128_r: use      = DEFLT
crt        = /lib/crt0.o
mcrtd      = /lib/mcrt0.o
gcrt       = /lib/gcrt0.o
libraries  = -L/usr/lib/threads,-lpthreads,-lc128,-lc
proflibs   = -L/lib/profiled,-L/usr/lib/profiled
hdlibs     = -L/usr/vac/lib,-lhmd
options    = -qlanglvl=extended,-qnor,-qnorconst,-qldb128,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__
* xlc C++ compiler aliased as xlc_r (AIX Threads)
xlc_r: use      = DEFLT
crt        = /lib/crt0.o
mcrtd      = /lib/mcrt0.o
gcrt       = /lib/gcrt0.o
options    = -qansialias,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__
libraries2 = -L/usr/lib/threads,-L/usr/vacpp/lib,-lc,-lpthreads,-lm,-lc
proflibs   = -L/usr/vacpp/lib/profiled,-L/lib/profiled,-L/usr/lib/profiled
hdlibs     = -L/usr/vac/lib,-lhC,-lhmd
* xlc C++ compiler with 128 bit long double aliased as xlc128_r (AIX Threads)
xlc128_r: use   = DEFLT
crt            = /lib/crt0.o
mcrtd          = /lib/mcrt0.o
gcrt           = /lib/gcrt0.o
libraries2     = -L/usr/lib/threads,-L/usr/vacpp/lib,-lc128,-lc,-lpthreads,-lc128,-lm,-lc
proflibs       = -L/usr/vacpp/lib/profiled,-L/lib/profiled,-L/usr/lib/profiled
hdlibs         = -L/usr/vac/lib,-lhC,-lhmd
options        = -qansialias,-qldb128,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__
* standard c compiler aliased as xlc_r4 (DCE)
xlc_r4: use    = DEFLT
crt           = /lib/crt0.o
mcrtd         = /lib/mcrt0.o
gcrt          = /lib/gcrt0.o
libraries     = -L/usr/lib/threads,-ldcelib_r,-ldcepthreads,-lpthreads_compat,-lpthreads,-lc
proflibs      = -L/lib/profiled,-L/usr/lib/profiled
hdlibs        = -L/usr/vac/lib,-lhmd
options       = -qansialias,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__,-D_AIX_PTHREADS_D7,-D_AIX32_THREADS=1,-D_AIX32_THREADS_D7
* xlc128_r4
xlc128_r4: use  = DEFLT
crt            = /lib/crt0.o
mcrtd          = /lib/mcrt0.o
gcrt           = /lib/gcrt0.o
libraries      = -L/usr/lib/threads,-ldcelib_r,-ldcepthreads,-lpthreads_compat,-lpthreads,-lc128,-lc
proflibs       = -L/lib/profiled,-L/usr/lib/profiled
hdlibs         = -L/usr/vac/lib,-lhmd
options        = -qansialias,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__,-D_AIX_PTHREADS_D7,-D_AIX32_THREADS=1,-D_AIX32_THREADS_D7
* standard c compiler aliased as cc_r4 (DCE)
cc_r4: use     = DEFLT
crt           = /lib/crt0.o
mcrtd         = /lib/mcrt0.o
gcrt          = /lib/gcrt0.o
libraries     = -L/usr/lib/threads,-ldcelib_r,-ldcepthreads,-lpthreads_compat,-lpthreads,-lc
proflibs      = -L/lib/profiled,-L/usr/lib/profiled
hdlibs        = -L/usr/vac/lib,-lhmd
options       = -qlanglvl=extended,-qnor,-qnorconst,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__,-D_AIX_PTHREADS_D7
* cc128_r4
cc128_r4: use  = DEFLT
crt            = /lib/crt0.o
mcrtd          = /lib/mcrt0.o
gcrt           = /lib/gcrt0.o
libraries      = -L/usr/lib/threads,-ldcelib_r,-ldcepthreads,-lpthreads_compat,-lpthreads,-lc128,-lc
proflibs       = -L/lib/profiled,-L/usr/lib/profiled

```

```

    hdlibs      = -L/usr/vac/lib,-lhmd
    options     = -qlanglvl=extended,-qnor,-qnorconst,-qthreaded,-D_THREAD_SAFE,-D_VACPP_MULTI__,-D_AIX_PTHREADS_D7,-
* x1C C++ compiler aliased as x1C_r4 (DCE)
x1C_r4: use    = DEFLT
    crt        = /lib/crt0.o
    mcrt       = /lib/mcrt0.o
    gcrt       = /lib/gcrt0.o
    libraries2 = -L/usr/lib/threads,-L/usr/vacpp/lib,-lc,-ldcelibc_r,-ldcepthreads,-lpthreads_compat,-lpthreads,-lm,-lc
    proflibs   = -L/usr/vacpp/lib/profiled,-L/lib/profiled,-L/usr/lib/profiled
    hdlibs     = -L/usr/vac/lib,-lhC,-lhmd
    options    = -qansialias,-qthreaded,-D_THREAD_SAFE,-D_VACPP_MULTI__,-D_AIX_PTHREADS_D7,-D_AIX32_THREADS=1,-D_AIX41
* x1C C++ compiler with 128 bit long double aliased as x1C128_r4 (DCE)
x1C128_r4: use = DEFLT
    crt        = /lib/crt0.o
    mcrt       = /lib/mcrt0.o
    gcrt       = /lib/gcrt0.o
    libraries2 = -L/usr/lib/threads,-L/usr/vacpp/lib,-lc128,-lc,-ldcelibc_r,-ldcepthreads,-lpthreads_compat,-lm,-lpthreads
    proflibs   = -L/usr/vacpp/lib/profiled,-L/lib/profiled,-L/usr/lib/profiled
    hdlibs     = -L/usr/vac/lib,-lhC,-lhmd
    options    = -qansialias,-qldb1128,-qthreaded,-D_THREAD_SAFE,-D_VACPP_MULTI__,-D_AIX_PTHREADS_D7,-D_AIX32_THREADS=
* standard c compiler aliased as acrt0r (AIX POSIX Draft 7Threads)
x1c_r7: use   = DEFLT
    crt       = /lib/crt0.o
    mcrt      = /lib/mcrt0.o
    gcrt      = /lib/gcrt0.o
    libraries = -L/usr/lib/threads,-lpthreads_compat,-lpthreads,-lc_r,-lc
    proflibs  = -L/lib/profiled,-L/usr/lib/profiled
    hdlibs    = -L/usr/vac/lib,-lhmd
    options   = -qansialias,-qthreaded,-D_THREAD_SAFE,-D_VACPP_MULTI__,-D_AIX_PTHREADS_D7
x1c128_r7: use = DEFLT
    crt       = /lib/crt0.o
    mcrt      = /lib/mcrt0.o
    gcrt      = /lib/gcrt0.o
    libraries = -L/usr/lib/threads,-lpthreads_compat,-lpthreads,-lc128,-lc_r,-lc
    proflibs  = -L/lib/profiled,-L/usr/lib/profiled
    hdlibs    = -L/usr/vac/lib,-lhmd
    options   = -qansialias,-qthreaded,-D_THREAD_SAFE,-D_VACPP_MULTI__,-D_AIX_PTHREADS_D7,-qldb1128
* standard c compiler aliased as cc_r (AIX POSIX Draft 7Threads)
cc_r7: use    = DEFLT
    crt       = /lib/crt0.o
    mcrt      = /lib/mcrt0.o
    gcrt      = /lib/gcrt0.o
    libraries = -L/usr/lib/threads,-lpthreads_compat,-lpthreads,-lc_r,-lc
    proflibs  = -L/lib/profiled,-L/usr/lib/profiled
    hdlibs    = -L/usr/vac/lib,-lhmd
    options   = -qlanglvl=extended,-qnor,-qnorconst,-qthreaded,-D_THREAD_SAFE,-D_VACPP_MULTI__,-D_AIX_PTHREADS_D7
cc128_r7: use = DEFLT
    crt       = /lib/crt0.o
    mcrt      = /lib/mcrt0.o
    gcrt      = /lib/gcrt0.o
    libraries = -L/usr/lib/threads,-lpthreads_compat,-lpthreads,-lc128,-lc_r,-lc
    proflibs  = -L/lib/profiled,-L/usr/lib/profiled
    hdlibs    = -L/usr/vac/lib,-lhmd
    options   = -qlanglvl=extended,-qnor,-qnorconst,-qthreaded,-D_THREAD_SAFE,-D_VACPP_MULTI__,-D_AIX_PTHREADS_D7,-
* x1C C++ compiler aliased as x1C_r7 (AIX POSIX Draft 7Threads)
x1C_r7: use   = DEFLT
    crt       = /lib/crt0.o
    mcrt      = /lib/mcrt0.o
    gcrt      = /lib/gcrt0.o
    libraries2 = -L/usr/lib/threads,-L/usr/vacpp/lib,-lc,-lpthreads_compat,-lpthreads,-lc_r,-lm,/usr/lib/libc.a
    proflibs   = -L/usr/vacpp/lib/profiled,-L/lib/profiled,-L/usr/lib/profiled
    hdlibs     = -L/usr/vac/lib,-lhC,-lhmd
    options    = -qansialias,-qthreaded,-D_THREAD_SAFE,-D_VACPP_MULTI__,-D_AIX_PTHREADS_D7
* x1C C++ compiler with 128 bit long double aliased as x1C128_r7 (AIX POSIX Draft 7Threads)
x1C128_r7: use = DEFLT
    crt       = /lib/crt0.o
    mcrt      = /lib/mcrt0.o

```

```

gcrt      = /lib/gcrt0.o
libraries2 = -L/usr/lib/threads,-L/usr/vacpp/lib,-lC128,-lC,-lpthreads_compat,-lm,-lpthreads,-lc128,-lc
proflibs  = -L/usr/vacpp/lib/profiled,-L/lib/profiled,-L/usr/lib/profiled
hdlibs    = -L/usr/vac/lib,-lhc,-lhmd
options   = -qansialias,-qldb1128,-qthreaded,-D_THREAD_SAFE,-D__VACPP_MULTI__, -D_AIX_PTHREADS_D7
* common definitions
DEFLT: cppcomp = /usr/vacpp/exe/xlCentry
      ccomp   = /usr/vac/exe/xlcentry
      code    = /usr/vac/exe/xlCcode
      cpp     = /usr/vac/exe/xlCcpp
      munch   = /usr/vacpp/exe/munch
      ipa     = /usr/vac/exe/ipa
      dis     = /usr/vac/exe/dis
      cppfilt = /usr/vacpp/bin/c++filt
      bolt    = /usr/vac/exe/bolt
      as      = /bin/as
      ld      = /bin/ld
      xlC     = /usr/vacpp/bin/xlC
      cppinc  = /usr/vacpp/include
      options = -D_AIX,-D_AIX32,-D_AIX41,-D_AIX43,-D_IBMR2,-D_POWER,-bpT:0x10000000,-bpD:0x20000000
      ldopt   = "b:o:e:u:R:H:Y:Z:L:T:A:k:j:"
      hdlibs  = -L/usr/vac/lib,-lhmd
      xlCcopt = -qansialias
      crt_64  = /lib/crt0_64.o
      mcrt_64 = /lib/mcrt0_64.o
      gcrt_64 = /lib/gcrt0_64.o
      smplibraries = -lxlsmplib
      optlibraries = -lxlopt
      resexp    = /usr/vacpp/lib/res.exp
      genexports = /usr/vac/bin/CreateExportList

```



## Appendix M. ASCII Character Set

The C for AIX compiler 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            |
| 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  |                         |
| 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 |                        |
| 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                   |

---

## Appendix N. Problem Solving

---

### 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. See “Chapter 2. Setting Up the C for AIX Compilation Environment” on page 3 for more information about setting these variables.

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 C for AIX 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/vac*.cat
```

where **%L** is the primary language environment (locale) selected during installation of the AIX Version 4 Operating System. If the AIX system has never been installed on the system before, the default locale is **C**. The locale for United States English is **en\_US**. The default message catalogs in **/usr/vac/exe/default\_msg** are called when:

- The C for AIX compiler cannot find message catalogs for the locale specified by **%L**.
- The locale has never been changed from the default, **C**.

For more information about the **NLSPATH** and **LANG** environment variables, see the *AIX Version 4 System User's Guide: Operating Systems and Devices*.

The AIX national language facilities are described in the “Introducing National Language Support for Programming” in the *AIX Version 4 General Programming Concepts*.

#### Related Reading

“Chapter 2. Setting Up the C for AIX Compilation Environment” on page 3

---

### Correcting Page Space Errors During Compilation

If the AIX 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**.

The paging-space overview in *AIX Version 4 System User's Guide: Operating System and Devices* describes paging space and how to allocate it.

---

## Appendix O. Glossary

This is a glossary of commonly used terms in the C for AIX information library. It includes definitions developed by the American National Standards Institute (ANSI) and entries from the *IBM Dictionary of Computing* (ZC20-1699). It supplements the *AIX Version 4 Topic Index and Glossary*.

---

### A

#### address

A name, label, or number identifying a location in storage, a device in a system or network, or any other data source.

#### arithmetic object

An integral object or objects having the type **float**, **double**, or **long double**.

#### array element

A single data item in an array.

#### assembler language

A symbolic programming language in which the set of instructions includes the instructions of the machine and whose data structures correspond directly to the storage and registers of the machine.

#### assignment conversion

A change to the form of the right operand that makes the right operand have the same data type as the left operand.

#### assignment expression

An operation that stores the value of the right operand in the storage location specified by the left operand.

---

### B

#### binary expression

An operation containing two operands and one operator.

#### bit field

A member of a structure or union that contains 0 or more bits.

#### block

See 489.

#### block statement

Any number of data definitions, declarations, and statements that appear between the symbols { (left brace) and } (right brace).

#### boundary alignment

The position in main storage of a fixed-length field (such as halfword or doubleword) on an integral boundary for that unit of information. For example, a word boundary is a storage address evenly divisible by four.

#### break statement

A language control statement that contains the word **break** and a semicolon. It is used to end an iterative or a switch statement by exiting from it at any point other than the logical end. Control is passed to the first statement after the iteration or switch statement.

#### buffer flush

A process that removes the contents of a buffer. After a buffer flush, the buffer is empty.

---

## C

### C library

A system library that contains common C language subroutines for file access, string operators, character operations, memory allocation, and other functions.

### case clause

In a switch statement, a case label followed by any number of statements.

### case label

The word **case** followed by a constant expression and a colon. When the selector evaluates the value of the constant expression, the statements following the case label are processed.

### cast expression

A cast expression explicitly converts its operand to a specified arithmetic, scalar, or class type.

### cast operator

The cast operator is used for explicit type conversions.

### char specifier

A char is a built-in data type.

### character constant

A character or an escape sequence enclosed in single quotation marks.

### character set

A group of characters used for a specific reason; for example, the set of characters a printer can print or a keyboard can support.

### character variable

A data object whose value can be changed during program execution and whose data type is **char**, **signed char**, or **unsigned char**.

### comma expression

An expression that contains two operands separated by a comma. Although the compiler evaluates both operands, the value of the expression is the value of the right operand. If the left operand produces a value, the compiler discards this value. Typically, the left operand of a comma expression is used to produce side effects.

### complex number

A complex number is made up of two parts: a real part and an imaginary part. A complex number can be represented by an ordered pair  $(a, b)$ , where  $a$  is the value of the real part and  $b$  is the value of the imaginary part. The same complex number could also be represented as  $a + bi$ , where  $i$  is the square root of  $-1$ .

### conditional compilation directive

A preprocessor directive that causes the preprocessor to process specified source code in the file depending on the evaluation of a specific condition.

### conditional expression

A compound expression that contains a condition (the first expression), an expression to be evaluated if the condition has a nonzero value (the second expression), and an expression to be evaluated if the condition has the value zero (the third expression).

**const** A keyword that allows you to define a variable whose value does not change.

### constant

A data item with a value that does not change during program execution.

### constant expression

An expression having a value that can be determined during compilation and that does not change during program execution.

**control statement**

A C language statement that changes the normal path of execution.

**conversion**

A change in the type of a value. For example, when you add values having different data types, the compiler converts both values to the same type before adding them.

---

**D****data definition**

A program statement that describes the features of, specifies relationships of, or establishes the context of, data. A data definition can also provide an initial value. Definitions appear outside a function (for example at file scope) or within a block statement.

**data object**

Anything that exists in storage and on which operations can be performed, such as files, programs, or arrays.

**data type**

A category that specifies the interpretation of a data object such as its mathematical qualities and internal representation.

**DBCS** See 491.

**debug** To detect, locate, and correct errors in a program.

**decimal constant**

A number containing any digits 0 through 9 that does not begin with 0 (zero).

**declaration**

Establishes the names and characteristics of data objects and functions used in a program.

**declarator**

Designates a data object or function declared. Initializations can be performed in a declarator.

**default**

A value, attribute, or option that is assumed when no alternative is specified by the programmer.

**default clause**

In a **switch** statement, the keyword **default** followed by a colon, and one or more statements. When the conditions of the specified **case** labels in the **switch** statement do not hold, the **default** clause is chosen.

**default initialization**

The initial value assigned to a data object by the compiler if no initial value is specified by the programmer. **extern** and **static** variables receive a default initialization of zero, while the default initial value for **auto** and **register** variables is undefined.

**define directive**

A preprocessor directive that causes the preprocessor to replace an identifier or macro call with specified code.

**definition**

A declaration that allocates storage, and may initialize a data object or specify the body of a function.

**do statement**

A looping statement that contains the word **do** followed by a statement (the action), the word **while**, and an expression in parentheses (the condition).

**double-byte character set ( DBCS)**

A set of characters in which each character is represented by 2 bytes of storage.

**double precision**

The use of two computer words to represent a floating-point value in accordance with the required precision.

---

**E****element**

The component of an array, subrange, enumeration, or set.

**else clause**

The part of an **if** statement that contains the word **else** followed by a statement. The else clause provides an action that is executed when the **if** condition evaluates to zero (false).

**enumeration constant**

An identifier (that has an associated integer value) defined by an enumeration type. You can use an enumeration constant anywhere an integer constant is allowed.

**enumeration tag**

The identifier that names an enumeration data type.

**enumeration type**

An enumeration type defines a set of enumeration constants.

**enumerator**

An enumeration constant and its associated value.

**escape sequence**

A representation of a nonprintable character in a character or string literal. An escape sequence contains the **\** symbol, followed by one of the characters: **a**, **b**, **f**, **n**, **r**, **t**, **v**, **'**, **?**, or **\**, or followed by one to three octal digits or **\** followed by an **x** followed by any number of hexadecimal digits.

**exception**

Any user, logic, or system error detected by a function that does not itself deal with the error but passes the error on to a handling routine.

**expression**

A representation of a value. For example, variables and constants appearing alone or in combination with operators are expressions.

**external data definition**

A definition appearing outside a function. The defined object is accessible to all functions that follow the definition and are located within the same source file as the definition.

---

**F****file scope**

A name declared outside all blocks and classes has file scope and can be used after the point of declaration in a source file.

**float constant**

A number containing a decimal point, an exponent, or both a decimal point and an exponent. The exponent contains an **e** or **E**, an optional sign (+ or -), and one or more digits (0 through 9).

**for statement**

A looping statement that contains the word **for** followed by a list of expressions enclosed in parentheses (the condition) and a statement (the action). Each expression in the parenthesized list is separated by a semicolon. You can omit any of the expressions, but you cannot omit the semicolons.

**function**

A named group of statements that can be invoked and evaluated and can return a value to the calling statement.



**function call**

An expression that moves the path of execution from the current function to a specified function and evaluates to the return value provided by the called function. A function call contains the name of the function to which control moves and a parenthesized list of arguments.

**function declarator**

The part of a function definition that names the function, provides additional information about the return value of the function, and lists the function parameters.

**function definition**

The complete description of a function. A function definition contains an optional storage class specifier, an optional type specifier, a function declarator, parameter declarations, and a block statement (the function body).

**function prototype**

A function declaration that provides type information for each parameter. It is the first line of the function (header) followed by a ; (semicolon). It is required by the compiler when the function will be declared later so type checking can occur.

**function scope**

Labels that are declared in a function have function scope and can be used anywhere in that function.

---

**G****global scope**

See 492.

**global variable**

A symbol defined in one program module that is used in other independently compiled program modules.

---

**H****header file**

A file that contains declarations used by a group of functions or users.

**hexadecimal**

A system of numbers to the base sixteen; hexadecimal digits range from 0 (zero) through 9 (nine) and A (ten) through F (fifteen).

**hexadecimal constant**

A constant, usually starting with special characters, that contains only hexadecimal digits. The special characters are `\x`, `0x`, or `0X`.

---

**I****I/O Stream Library**

A class library that provides the facilities to deal with many varieties of input and output.

**identifier**

A name that refers to a data object. An identifier contains some combination of letters, digits, and underscores, but its first character cannot be a digit.

**if statement**

A conditional statement that contains the word **if** followed by an expression in parentheses (the condition), a statement (the action), and an optional else clause (the alternative action).

**include file**

A text file that contains declarations used by a group of functions, programs, or users. Also known as a header file.

**include directive**

A preprocessor directive that causes the preprocessor to replace the statement with the contents of a specified file.

**initialize**

To set the starting value of a data object.

**initializer**

An expression used to initialize data objects.

**inline function**

A function declared and defined simultaneously in a class definition. You can also explicitly declare a function inline by using the keyword **inline**, which is a hint to the compiler to perform inline expansion of the body of a function member. Both member and nonmember functions can be inlined.

**instruction**

A program statement that specifies an operation to be performed by the computer, along with the values or locations of operands. This statement represents the programmer's request to the processor to perform a specific operation.

**integer**

A positive or negative whole number or zero.

**integer constant**

A decimal, octal, or hexadecimal constant.

**integral object**

A character object, an object having variations of the type **int**, or an object that is a bit field.

**internal data definition**

A description of a variable appearing in a block that directs the system to allocate storage for that variable and makes that variable accessible to the current block after its point of declaration.

---

**K****keyword**

A reserved C language identifier.

---

**L**

**label** An identifier followed by a colon, used to identify a statement in a program. Usually the target of a **goto** or **switch** statement.

**labeled statement**

A possibly empty statement immediately preceded by a label.

**link** To interconnect items of data or portions of one or more computer programs, such as linking object programs by a linkage editor or linking data items by pointers.

**linkage editor**

A program that resolves cross-references between separately compiled object modules and then assigns final addresses to create a single relocatable load module. If a single object module is linked, the linkage editor simply makes it relocatable.

**literal** See 490.

**load module**

A computer program in a form suitable for loading into main storage for execution.

**local** Pertaining to information that is defined and used only in one subdivision of a computer program.

**local scope**

A name declared in a block has local scope and can only be used in that block.

**long constant**

An integer constant followed by the letter **I** (el) or **L**.

**lvalue** An expression that represents an object. A modifiable **lvalue** can be both examined and changed.

---

## M

**macro call**

An identifier followed by a parenthetical list of arguments that the preprocessor replaces with the replacement code located in a preprocessor **define** directive.

**main function**

An external function that has the identifier **main**. Each program must have exactly one external function named **main( )**. Program execution begins with this function.

---

## N

**new-line character**

A control character that causes the print or display position to move to the first position on the next line. This character is represented by '\n' in C.

**NULL** A pointer that has a value 0 is guaranteed not to point to any data object. The pointer can be converted to any pointer type.

**null character (NUL)**

The character hex 00, used to represent the absence of a printed or displayed character.

**null statement**

A C statement that consists solely of a semicolon.

---

## O

**object code**

Machine-executable instructions, usually generated by a compiler from source code written in a higher level language. For programs that must be linked, object code consists of relocatable machine code.

**octal** A base eight numbering system.

**octal constant**

The digit 0 (zero) followed by any digits 0 through 7.

**operand**

An entity on which an operation is performed.

**operator**

A symbol (such as +, -, \*) that represents an operation (in this case, addition, subtraction, multiplication).

**overflow**

That portion of an operation's result that exceeds the capacity of the intended unit of storage.

**overflow condition**

A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

---

## P

**pad** To fill unused positions in a field with dummy data, usually zeros, ones, or blanks.

**parameter declaration**

A description of a value that a function receives. A parameter declaration determines the storage class and the data type of the value.

**pointer**

A variable that holds the address of a data object or function.

**preprocessor**

A program that examines the source program for preprocessor statements that are then interpreted, resulting in the alteration of the source program.

**preprocessor directive**

A statement that begins with the pound sign (#) and contains instructions that the preprocessor interprets.

**primary expression**

Literals, names, and names qualified by the :: (scope resolution) operator.

**prototype**

See 492.

---

## R

**register**

A storage area commonly associated with fast-access storage, capable of storing a specified amount of data such as a bit or an address.

**register variable**

A variable defined with the **register** storage class specifier. Register variables have automatic storage.

---

## S

**scalar** An arithmetic object, or a pointer, or a reference to an object of any type.

**scope** That part of a source program in which a variable is visible.

**single-byte character set**

A set of characters in which each character is represented by 1 byte of storage.

**single precision**

Pertaining to the use of one computer word to represent a number, in accordance with the required precision.

**source program**

A set of instructions written in a programming language that must be translated to machine language before the program can be run.

**stanza**

A group of lines in a file that together have a common function or define a part of a system. Stanzas are usually separated by blank lines or colons, and each stanza has a name. The stanzas of the C for AIX compiler default configuration file, **/etc/vac.cfg**, specify information that the compiler uses when it is invoked.

**statement**

An instruction that ends with a semicolon (;) or one or more instructions enclosed in braces ({}).

**static** A keyword used for defining the scope and linkage of variables and functions. For internal variables, the variable has block scope and retains its value between function calls. For external values, the variable has file scope and retains its value within the source file. For class variables, the variable is shared by all objects of the class and retains its value within the entire program.

**storage class specifier**

One of: **auto**, **register**, **static**, or **extern**.

**string literal**

Zero or more characters enclosed in double quotation marks.

**structure**

A data type that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types.

**structure tag**

The identifier that names a structure data type.

**subscript**

One or more expressions, each enclosed in brackets, that follow an array name. A subscript refers to an element in an array.

**switch expression**

The controlling expression of a **switch** statement.

**switch statement**

A C language statement that causes control to be transferred to one of several statements depending on the value of an expression.

---

## T

**token** The smallest independent unit of meaning of a program as defined either by a parser or a lexical analyzer. A token can contain data, a language keyword, an identifier, or other parts of language syntax.

**translation**

The process of transforming all or part of a source program into a program image that contains all the information needed for the program to run.

**type** The description of the data and the operations that can be performed on or by the data.

**type balancing**

A conversion that makes both operands have the same data type. If the operands do not have the same size data type, the compiler converts the value of the operand with the smaller type to a value having the larger type.

**type conversion**

See 489.

**type definition**

A definition of a data type.

**type specifier**

Used to indicate the data type of an object or function being declared.

---

## U

**ultimate consumer**

The target of data in an input and output operation. An ultimate consumer can be a file, a device, or an array of bytes in memory.

**ultimate producer**

The source of data in an input and output operation. An ultimate producer can be a file, a device, or an array of bytes in memory.

**unary expression**

An expression that contains one operand.

**union** A variable that can hold any one of several data types, but only one data type at a time.

**union tag**

The identifier that names a union data type.

---

**V****variable**

An object that can take different values at different times.

---

**W****while statement**

A looping statement that contains the word **while** followed by an expression in parentheses (the condition) and a statement (the action).

**white space**

Space characters, tab characters, form feed characters, new-line characters, and (when referring to source code) comments.

---

**Z****zero suppression**

The removal of, or substitution of blanks for, leading zeros in a number. For example, 00057 becomes 57 when using zero suppression.



**IBM**